UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Gestione delle transazioni in MongoDB

Progetto di New Generation Databases

Studentesse:
Parlapiano Federica
Ronci Arianna

Docente:
Prof.ssa Diamantini Claudia

Anno Accademico 2023-2024

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Gestione delle transazioni in MongoDB

Progetto di New Generation Databases

Studentesse:
Parlapiano Federica
Ronci Arianna

Docente:
Prof.ssa Diamantini Claudia

Anno Accademico 2023-2024

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Indice

Capitolo 1

Le transazioni in MongoDB

1.1 Transazione in MongoDB

In MongoDB un'operazione su un singolo documento è atomica. Grazie al fatto che non si usa la normalizzazione, l'atomicità su un singolo documento rende superfluo il ricorso a transazioni distribuite in molti casi.

In quelle situazioni in cui è richiesta atomicità per le letture e per le scritture di più documenti (in una singola o in più collezioni), MongoDB supporta le transazioni distribuite.

Le transazioni sono atomiche:

- le transazioni applicano tutte le modifiche ai dati o le annullano;
- l'esecuzione di una transazione comporta che tutte le modifiche ai dati apportate dalla transazione vengono salvate e rese visibili al di fuori della transazione stessa;
- prima del commit della transazione le modifiche ai dati apportate dalla transazione non sono visibili al di fuori della transazione stessa;
- se la transazione scrive su più shard, non è necessario che le operazioni di lettura esterne attendano che il risultato della transazione andata in commit sia visibile su tutti gli shard; per esempio, se una transazione effettua due scritture su due shard e va in commit, potrebbe accadere che, nel momento in cui si esegue una lettura esterna con read concern *local*, la scrittura 1 sia visibile sullo shard A, mentre la scrittura 2 non sia ancora visibile sullo shard B; in questo modo sono letti i risultati della scrittura 1 senza vedere la scrittura 2;
- se una transazione va in abort, tutte le modifiche apportate ai dati nella transazione vengono scartate senza diventare mai visibili.

Le transazioni distribuite possono essere utilizzate per più operazioni, collezioni, database, documenti e shard.

In una transazione è possibile creare collezioni e indici. Le collezioni utilizzate in una transazione possono trovarsi anche in database diversi.

Una transazione è associata a una sessione. Ogni sessione può avere al massimo una transazione in corso alla volta. Se viene chiusa una sessione e in tale sessione c'è una transazione in corso, questa viene mandata in abort.

Quando si mandano in esecuzione delle transazioni è possibile specificare read concern, write concern e read preference.
L'uso corretto ed efficace di questi parametri permette di regolare il livello di garanzia di consistenza e la disponibilità dei dati.

1.1.1 Transactions API

L'API Transactions poggia sulla Callback API.
La Callback API inizia una transazione, ne esegue le operazioni e il commit o, qualora andasse in errore, l'abort.
Inoltre, include la logica per la gestione degli errori $TransientTransactionError$ e $UnknownTransactionCommitResult$.

1.1.2 Tempo di vita delle transazioni

Le transazioni hanno un tempo di vita, trascorso il quale vengono mandate in abort da un periodico processo di pulizia.
Di default, tale tempo di vita corrisponde a 60 secondi. Si può modificare usando il seguente comando:

```
db.adminCommand( { setParameter: 1, transactionLifetimeLimitSeconds: 30 } )
```

Listing 1.1: Modificare il tempo di vita di una transazione

A livello di sessione si può modificare in questo modo:

```
const session = db.getMongo().startSession({
    defaultTransactionOptions: {
        readConcern: { level: "majority" },
        writeConcern: { w: "majority" },
        transactionLifetimeLimitSeconds: 20
    }
});
```

1.2 Read Preference

Con Read Preference si intende l'opzione che in MongoDB permette di specificare la preferenza di lettura. Quando si esegue un'operazione di lettura, per impostazione predefinita, questa è indirizzata al nodo primario del Replica Set, che è l'unico a fornire garanzie sull'aggiornamento dei dati letti. Per modificare tale comportamento predefinito in lettura è possibile specificare una preferenza di lettura attraverso

appunto la proprietà nota come *ReadPreference*. In questo modo un client può inviare le operazioni di lettura ai membri secondari.

Se questo è vero in generale, quando si tratta di transazioni, il comportamento della read preference cambia. Per garantire che i dati letti siano coerenti con quelli scritti nella stessa transazione, tutte le operazioni di una determinata transazione devono essere indirizzate allo stesso membro del Replica Set. Questa considerazione, insieme al fatto che il nodo primario è l'unico che può eseguire operazioni di scrittura, implica che una transazione necessariamente legge dal nodo primario. Per una transazione si deve sempre usare la read preference *primary*. Una modalità differente determina una condizione di errore.
Inoltre, anche se è possibile specificare una read preference a livello di sessione o di collezione, durante una transazione questa viene ignorata e le letture vengono forzate al primary.

Nella definizione della read preference si possono indicare la *read preference mode*, una *tag set list*, le opzioni maxStalenessSeconds e hedged read.

1.2.1 Read Preference Mode

I possibili valori per la *read preference mode* sono i seguenti.

- *primary*: è il valore di default ed indica che tutte le operazioni di lettura vengono eseguite sul nodo primario. Le operazioni in una transazione devono essere indirizzate tutte allo stesso membro, per cui le transazioni che includono operazioni di lettura devono usare la read preference primary.

- *primaryPreferred*: sebbene spesso le operazioni leggono dal nodo primario, se questo non è disponibile, le operazioni eseguono la lettura dai secondari.
  Quando si specifica un valore per il parametro maxStalenessSeconds e non si può leggere dal primario, il client stima quanto non sono aggiornati i dati di ciascun secondario confrontando l'ultima scrittura del secondario con quella del secondario con la scrittura più recente. Il client dopo indirizza la lettura al secondario il cui ritardo è minore o uguale a maxStalenessSeconds.
  Se si include una tag set list e non si può leggere dal primario, il client cerca i secondari che corrispondono a un tra i tag set indicati, procedendo in ordine fino a che non si trova una corrispondenza con un tag set. Se si trova, il client sceglie randomicament dal set e un nodo da cui leggere. Altrimenti, viene lanciato un errore.
  Specificando sia un valore per maxStalenessSeconds che una tag list, prima si filtra per maxStalenessSeconds e poi per la tag list.

- *secondary*: tutte le operazioni eseguono le letture sui membri secondary. Se nessuno dei nodi secondari è disponibile, l'operazione di lettura produce un errore. Solitamente un Replica Set ha almeno un secondario, ma potrebbe succedere che non ci siano nodi secondari disponibili. Se nella read preference si specifica anche la tag set list, il client cerca il membro secondario che corrisponde ad un tag. Se non viene trovata nessuna corrispondenza, l'operazione di lettura produce un errore.
  Se viene indicato un valore per il maxStalenessSeconds, il client stima quanto non è aggiornato ogni nodo secondario confrontando l'ultima scrittura del nodo secondario con quella del primario. Il client indirizza l'operazione di lettura al nodo secondario che ha un ritardo minore o uguale di maxStalenessSeconds. Se non c'è un primario, per fare il confronto, il client usa il secondario con la scrittura più recente.
  Se vengono specificati sia un valore per maxStalenessSeconds che una tag list, prima si filtra per maxStalenessSeconds e poi per la tag list.

- *secondaryPreferred*: le letture avvengono tipicamente sui nodi secondari. Se il Replica Set ha solo un nodo, il primario , allora le letture avvengono sul primario.
  Il comportamento che si verifica specificando i parametri maxStalenessSeconds e tag set list è uguale a quello specificato nel punto precedente.

- *nearest*: i dati vengono letti da un nodo casuale, indipendentemente da se esso sia primario o secondario, basandosi solo su una soglia di latenza. Specificando un valore per maxStalenessSeconds, il comportamento è analogo ai precedenti, con la differenza che, una volta filtrati i nodi con valore maggiore di quello indicato, la richiesta del client viene casualmente indirizzata a un membro la cui latenza rientra nella finestra di latenza accettabile.
  Specificando il tag set list il comportamento è analogo al caso precedente.
  Se si specificano sia il maxStalenessSeconds e la tag set list, prima viene filtrato il risultato per maxStalenessSeconds e poi per i tag. Una volta applicato il filtro, si sceglie tra le rimanenti istanze il nodo la cui latenza rientra in quella accettabile.

Ad eccezione della read preference *primary*, le altre read preference possono restituire dati non aggiornati perché la replicazione dei dati sui nodi secondari è asincrona. Indicare una certa read preference non ha effetti sulla visibilità dei dati né sulla causal consistency.

1.2.2 maxStalenessSeconds

A causa di congestioni della rete, basso throughput del disco, operazioni che impiegano molto tempo, può esserci del ritardo tra i nodi secondari e il primario. L'opzione maxStalenessSeconds permette di specificare un massimo ritardo di replicazione (o

staleness) per le letture dal secondario. Se il ritardo stimato sul secondario eccede il valore indicato per l'opzione maxStalenessSeconds, il client ne interrompe l'uso per le operazioni di lettura.

Questa opzione può esser specificata se si è indicato primaryPreferred, secondary, secondaryPreferred o nearest come read preference mode.

Non c'è un maxStalenessSeconds di default, quindi i client non considerano il ritardo di un secondario quando scelgono dove indirizzare la lettura.

Il valore minimo che si può indicare è 90 secondi, altrimenti viene lanciato un errore. Tale valore minimo è giustificato dal fatto che i client stimano quanto i dati non sono aggiornati andando a verificare periodicamente l'ultima scrittura di ogni membro del Replica Set e, siccome questi controlli sono poco frequenti, la stima è approssimata.

1.2.3 Tag set list

La *tag set list* consente di specificare un tag set per indirizzare le operazioni ai membri (o membro) del Replica Set con i tag specificati.

La ricerca è fatta in successione fino a che una corrispondenza non viene trovata.
Una volta trovata la corrispondenza, i restanti tag set non vengono considerati.
Se non viene trovata, l'operazione termina con un errore.

1.2.4 Hedged read

Attraverso il parametro *hedged read* si può indicare l'uso di hedged read sui cluster sharded.
In questo modo le operazioni di lettura vengono instradate a due membri del set di replica per ogni shard interrogato e restituiscono i risultati da quello che risponde per primo per ogni shard.

1.3 Read Concern

In MongoDB, l'opzione *readConcern* permette di specificare il livello della consistenza e dell'isolamento delle operazione di lettura e dei dati letti dai Replica Set e dai cluster sharded.

I livelli di *readconcern* che possono essere specificati sono i seguenti:

- *local*,
- *available*,

- *majority*,
- *linearizable*,
- *snapshot*.

Per un operazione di lettura, il livello *local* restituisce i dati presenti sulla singola istanza alla quale la query è rivolta, senza garanzia che questi siano replicati ad altri nodi. Tale modalità non dà garanzia che i dati letti siano stati scritti dalla maggioranza dei membri del Replica Set, per cui la query può restituire dati che sono stati scritti ma non ancora confermati e che potrebbero quindi andare incontro ad un rollback. Inoltre, i dati più recenti per un nodo non è detto che siano quelli più recenti per l'intero sistema.
$ReadConcern(level = \text{“}local\text{''})$ è il valore predefinito per le operazioni di lettura sia sul nodo primario che sui nodi secondari.

Il livello *available* in un cluster unsharded ha lo stesso effetto che si avrebbe con *local*: l'operazione di lettura restituisce i dati dell'istanza senza alcuna garanzia che questi siano stati scritti sulla maggior parte dei membri del Replica Set. In altre parole, garantisce che i dati letti siano quelli disponibili localmente, senza preoccuparsi della sincronizzazione con gli altri nodi.
Se, invece, il cluster è sharded, garantisce una maggiore tolleranza in caso di partizioni di rete. Infatti, nel caso in cui si verificasse una partizione di rete, le query verrebbero ridirezionate sullo shard che si sarebbe scelto prima che la partizione si verificasse. Inoltre, non richiede di attendere garanzie di consistenza dai server che potrebbero non essere disponibili a causa della partizione.
Come nel caso del *local*, i dati più recenti in un nodo potrebbero non corrispondere a quelli più recenti nell'intero sistema.

Il livello *majority* garantisce che i dati letti siano stati scritti dalla maggior parte dei membri del Replica Set.
Nel caso di operazioni multi-documento, questo è garantito solo se la transazione fa il commit usando il writeConcern *majority*, altrimenti non dà nessuna garanzia.
Per soddisfare il read concern *majority*, ogni membro del Replica Set mantiene in memoria una vista dei dati in corrispondenza del majority-commit point, ossia il punto di conferma della maggioranza, calcolato dal nodo primario; il nodo interessato dall'operazione di lettura restituisce i dati a partire da questa vista.

Con il read concern *linearizable*, una query restituisce dati che riflettono le scritture confermate dalla maggioranza e completate prima dell'inizio dell'operazione di lettura stessa. A tal proposito, la query potrebbe attendere che le scritture contemporanee si siano propagate alla maggioranza dei membri del Replica Set prima di restituire i risultati.

Linearizable è il livello di read concern che assicura che le letture siano fortemente consistenti: quando un'operazione di lettura viene eseguita con read concern *linearizable*, MongoDB garantisce che la lettura restituirà l'ultimo stato scritto con successo, riflettendo la realtà coerente dei dati più recente e assicurando che l'operazione di lettura non restituirà una versione obsoleta del documento.
Si può specificare il read concern *linearizable* solo per le operazioni di lettura sul nodo primario. Inoltre, le garanzie date dal read concern *linearizable* si applicano solo se le operazioni di lettura interessano un singolo documento.
Con un read concern *linearizable* è sempre buona norma utilizzare il metodo $maxTimeMS$ per fare in modo che, nel caso in cui la maggioranza dei membri portanti di dati sia non disponibile, l'operazione restituisca un errore per indicare che il read concern non può essere soddisfatto, senza che questo determini un blocco. MongoDB termina le operazioni che superano il loro limite di tempo assegnato utilizzando il meccanismo di $db.killOp()$.

```
db.collection.find(
   { <query> },
   { <projection> }
).maxTimeMS( <milliseconds> )
```

Listing 1.2: Indicare il maxTimeMS

Il read concern *snapshot* assicura che una query restituisca dati che sono stati confermati dalla maggioranza dei membri del Replica Set fino a un punto specifico nel passato. Ciò significa che le letture sono effettuate su uno snapshot, un'istantanea dei dati che riflette lo stato dei dati in un momento immediatamente precedente a quello in cui la lettura è stata avviata, garantendo coerenza nel contesto della transazione. Questo è utile in scenari in cui si ha bisogno di letture consistenti, specialmente durante transazioni multi-documento, all'interno delle quali il read concern *snapshot* assicura che tutte le letture di una transazione vedano i dati come apparivano al momento dell'inizio della transazione stessa. Le letture non vengono influenzate da eventuali scritture simultanee effettuate in altri contesti o transazioni.
Questo valore fornisce le sue garanzie solo se la transazione scrive con writeConcern *majority*.

1.4 Write Concern

writeConcern è un'opzione che determina il livello di garanzia richiesto per le operazioni di scrittura, ovvero specifica quanto MongoDB deve assicurarsi che una scrittura sia stata applicata e replicata prima di considerarla completata con successo. Tale parametro aiuta a bilanciare la consistenza e la disponibilità delle operazioni di scrittura.
Si può specificare un write concern a livello di transazione.
Le singole operazioni di una transazione fanno riferimento al write concern della transazione in cui sono contenute. Non è però possibile specificare un write concern

per un'operazione specifica di una transazione. Se si specifica un write concern per una specifica operazione, viene lanciato un errore.

Al momento dell'inizio della transazione è possibile indicare il livello di write concern che si desidera utilizzare. Se non viene specificato, la transazione eredita il write concern dalla sessione. Se neanche a livello di sessione è indicato, si usa quello di default.

Il write concern può essere indicato specificando i seguenti campi:

```
1 { w: <value>, j: <boolean>, wtimeout: <number> }
```

Listing 1.3: Indicare il write concern

L'opzione *w* permette di richiedere la conferma che l'operazione di scrittura sia stata propagata a uno specifico numero di istanze o alle istanze di cui viene indicata l'etichetta.
L'opzione *j* permette di specificare se si voglia la conferma che l'operazione di scrittura sia stata registrata sul journal.
L'ultima opzione, *wtimeout*, è utile per prevenire le situazioni di blocco indefinito.

1.4.1 w

Come precedentemente accennato, l'opzione *w* permette di indicare la conferma che l'operazione di scrittura sia stata propagata a un certo numero di nodi del Replica Set.

I valori attribuibili a *w* sono i seguenti.

- *majority* richiede che le operazioni di scrittura siano state propagate ad almeno la maggioranza dei nodi votanti. Una volta datagli la conferma, il client può leggere il risultato di quella scrittura con un read concern *majority*.
  Un caso particolare è quello che si verifica nell'ambito delle transazioni multi-documento. Se la transazione non riesce a replicare i dati alla maggioranza dei nodi del Replica Set, potrebbe non andare immediatamente in rollback. Tuttavia, prima o poi il Replica Set sarà consistente (si parla di eventual consistency).

- Si può specificare un numero intero, che indica il numero di istanze a cui la modifica deve esser propagata.
  Casi particolari sono il numero 1 e 0.
  Il primo valore impone che la scrittura sia stata propagata all'istanza standalone di mongod o, nel caso di Replica Set, al nodo primario.
  Indicando 0, non si richiede nessuna conferma.

Un valore maggiore di 1 richiede la conferma dal primario e da tanti secondari quanti servono per raggiungere il valore indicato dal write concern.

- Infine, è possibile specificare un write concern personalizzato, che richiede che la scrittura sia stata propagata ai membri di cui sono indicate le etichette nel documento *settings.getLastErrorModes*.

Se non viene specificato nessun valore, quello di default è *majority*.

1.4.2 j

j è l'opzione che permette di richiedere la conferma che l'operazione di scrittura sia stata scritta sul journal su disco.
I valori che può assumere sono *true* e *false*.

Specificando true, si richiede la conferma che le istanze o il numero di istanze indicate dal valore dato a *w* abbiano scritto sul journal.
Si ricorda che il journaling è una procedura che garantisce durabilità in casi di failure.

Nel caso di un'istanza MongoDB standalone, l'acknowledgment delle operazioni di scrittura può esser fatto o dopo che è stata scritta in memoria o dopo che è stata scritta nel journal su disco. Questo dipende dal valore di *w* e di *j*.
Se *j* non viene specificato:

- se *w:1*, allora l'ack viene fatto dopo che la scrittura è stata fatta su disco;
- se *w:"majority"*, l'ack avviene dopo che l'operazione è stata scritta sul journal nel disco.

Se *j:true*, viene fatto l'ack dopo che l'operazione è stata registrata sul journal.
Se *j:false*, la conferma viene dopo che l'operazione di scrittura è avvenuta in memoria.

In presenza di replica set, se *j* non viene specificato l'acknowledgment dipende dal valore di *writeConcernMajorityJournalDefault*. Se è *true*, implica che il valore di *j* sia *true*. Se *false*, il valore di *j* è di conseguenza *false*, che quindi per l'acknowledgment richiede che l'operazione di scrittura sia stata fatta in memoria.

1.4.3 wtimeout

L'opzione *wtimeout* permette di indicare un tempo in millisecondi che indica il limite per ottenere il write concern. Può essere indicato solo se si è scelto un valore più grande di 1 per *w*.
Superato il tempo specificato, l'operazione di scrittura termina con un errore.
Se non si specifica un tempo limite, si rischia di bloccare indefinitamente l'operazione di scrittura. Infatti, se il livello di write concern non è raggiungibile, l'operazione si

blocca.
Indicare il valore 0 equivale a non indicare nessun valore.

Anche quando l'operazione di scrittura ha successo, se il tempo limite viene superato, viene lanciato un errore.
In queste situazioni, nel momento in cui l'operazione di scrittura fa il return, non vengono disfatte le modifiche effettuate prima che il tempo limite venisse superato.

1.5 Read e write concern per una transazione

Si può specificare l'isolamento a livello di singola transazione come segue.

```
1 const transactionOptions = { readConcern: { level: 'local' }, writeConcern: { w: 'majority' },
      readPreference: { mode: 'primary' } };
2 session.startTransaction(transactionOptions);
```

Listing 1.4: Indicare il Write Concern

Per i Replica Set e i cluster sharded si possono impostare dei criteri di lettura predefiniti globali, che vengono ereditati dalle operazioni per le quali non è specificato esplicitamente un *readConcern*. Il comando che permette di impostare i criteri globali è $setDefaultRWConcern$. Tale comando deve essere eseguito sul database di amministrazione.

```
1 db.adminCommand(
2   {
3     setDefaultRWConcern : 1,
4     defaultReadConcern: { <read concern> },
5     defaultWriteConcern: { <write concern> },
6     writeConcern: { <write concern> },
7     comment: <any>
8   }
9 )
```

Listing 1.5: Comando per impostare read e write concern globali

Il campo *setDefaultRWConcern* è un intero settato a 1.
Il campo *defaultReadConcern* è un documento che contiene la configurazione globale per il readConcern. Possono esser specificati i livelli *local, available, majority*. Il campo *defaultWriteConcern* è un documento che contiene la configurazione globale per il writeConcern. Possono esser specificati tutti i valori possibili per il writeConcern, ad eccezione di $w : 0$.
writeConcern è un parametro opzionale che consente di specificare il writeConcern che il comando *setDefaultRWConcern* deve usare.

Capitolo 2

Gestione della concorrenza in MongoDB

2.1 Meccanismo di locking

MongoDB utilizza un sistema di *lock a granularità multipla* che consente alle operazioni di bloccare altre operazioni a livello globale, a livello di database o a livello di collezione; inoltre, permette ai singoli motori di storage di implementare un proprio controllo della concorrenza ad un livello inferiore di quello riferito ad una collezione (ad esempio, a livello di documento in WiredTiger).
In particolare, MongoDB utilizza reader-writer lock per le gestione di accessi concorrenti a una risorsa, come un database o una collezione. Le modalità di locking previste sono:

- locking condiviso (S) per le letture,
- locking esclusivo (X) per le operazioni di scrittura,
- locking intent shared (IS) che indica un intento di leggere una risorsa a un livello inferiore,
- locking intent exclusive (IX) che indica un intento di scrivere una risorsa a un livello inferiore.

Lo scopo dei lock di intent è quello di, quando si blocca una risorsa ad una certa granularità, bloccare intenzionalmente tutti i livelli superiori. Ad esempio, quando si blocca una collezione per la scrittura, utilizzando un lock esclusivo (X), sia a livello di database a cui la collezione appartiene che a livello globale viene dichiarato un lock intent exclusive (IX).

Un lock esclusivo (X), in quanto tale, blocca tutte le altre operazioni fino a quando non viene rilasciato e quindi non può coesistere con nessun altro lock. Un lock condiviso (S) permette a più operazioni di leggere la risorsa contemporaneamente e quindi può coesistere solo con altri lock condivisi o lock intent shared (IS). Le modalità di lock IS e IX possono coesistere essendo che al livello corrente non determinano conflitti.

Le varie richieste di lock da parte delle operazioni sono accodate, nell'ordine in cui arrivano.

Per ottimizzare il throughput, quando una richiesta di lock viene concessa, tutte le altre richieste di lock compatibili vengono concesse contemporaneamente, potenzialmente rilasciando i lock prima che una richiesta di lock in conflitto venga eseguita. Stante tale meccanismo di base, è tuttavia possibile che in determinati scenari un'operazione, anche se non terminata, ceda il proprio lock per permettere ad un'altra di procedere; questo accade ad esempio per consentire le operazioni che richiedono l'accesso esclusivo a un insieme di dati, come la cancellazione e la creazione di indici/collezioni.

A partire da MongoDB 5.0, alcune operazioni di lettura non sono bloccate quando un'altra operazione detiene un blocco di scrittura esclusivo (X) sulla medesima risorsa. Queste operazioni sono: $find$, $count$, $distinct$, $aggregate$, $mapReduce$, $listCollections$, $listIndexes$.

2.2 Concorrenza in un Replica Set

Con i replica set, quando MongoDB scrive in una collezione sul nodo primario, scrive anche nel suo *oplog*, che è una collezione speciale nel database locale, che mantiene un registro di tutte le operazioni che modificano i dati. Di conseguenza, il nodo primario del Replica Set deve bloccare contemporaneamente sia il database della collezione sia il database locale; ciò è necessario per mantenere la consistenza del database e garantire che le operazioni di scrittura, anche con la replica, siano operazioni atomiche.
In fase di replica poi, quando dal primario le modifiche si propagano sui nodi secondari, MongoDB non applica le scritture in modo seriale sui secondari. I secondari raccolgono le oplog entries in blocchi e poi applicano questi blocchi in parallelo. Le scritture vengono applicate nell'ordine in cui appaiono nell'oplog.

Le letture che hanno come target i secondari leggono da uno snapshot dei dati se il secondario sta subendo un'operazione di replica. Questo permette alla lettura di avvenire contemporaneamente alla replica, garantendo comunque una visione coerente dei dati. In altre parole, anche se il nodo secondario sta ricevendo aggiornamenti in tempo reale dal nodo primario attraverso il processo di replica, le letture non vengono bloccate. Questo approccio evita che le letture siano rallentate dalle operazioni di replica. Tuttavia, questo non garantisce che i dati letti siano la versione più recente poiché questo dipende dal livello di isolamento specificato attraverso il parametro read concern.

2.3 Concorrenza e transazioni

Per gestire transazioni concorrenti che operano sugli stessi documenti, MongoDB combina il meccanismo di controllo della concorrenza a livello di documento con il meccanismo di controllo delle versioni.
Con il locking a livello di documento, quando una transazione modifica un documento è garantito che solo una transazione alla volta possa modificare quel documento.
Con il controllo delle versioni, invece, MongoDB è in grado di garantire che le modifiche a un documento siano coerenti. In questo modo, se più transazioni tentano di modificare lo stesso documento contemporaneamente, solo una delle transazioni può avere successo. Le altre transazioni riceveranno un errore di conflitto di scrittura.
Per individuare tali conflitti, durante una transazione, MongoDB tiene traccia delle versioni dei documenti. Se una transazione tenta di scrivere su un documento che è stato modificato da un'altra transazione, si verifica un conflitto.
In caso di conflitto, MongoDB annulla (rollback) le operazioni non riuscite e restituisce un errore. A livello di applicazione si può quindi gestire il conflitto riprovando la transazione.

Durante una transazione in MongoDB, le letture utilizzano uno snapshot isolato dei dati, il che significa che le operazioni di lettura vedono solo i dati che erano presenti all'inizio della transazione, più eventuali modifiche apportate dalla transazione stessa, ma non sono in grado di osservare eventuali cambiamenti apportati da altre transazioni ed operazioni concorrenti, in corso o concluse durante la transazione stessa. Questo comportamento è tale, indipendentemente dal livello di read concern specificato per la transazione, che influenza semplicemente quali dati sono inclusi nello snapshot, determinando la coerenza e l'isolamento delle letture durante la transazione.
La creazione dello snapshot in una transazione avviene al momento della prima operazione di lettura o di scrittura effettuata, piuttosto che all'inizio della transazione.
Nel dettaglio:

1. con il comando $startTransaction()$ viene iniziata la transazione; il sistema configura il contesto della transazione ma non crea immediatamente l'istantanea dei dati;

2. alla prima operazione che legge o modifica i dati nel contesto della transazione viene creato uno snapshot dei dati che rappresenta lo stato del database nel momento in cui questa operazione viene eseguita, inclusi tutti i dati già committati fino a quel momento;

3. tutte le letture successive all'interno della transazione si basano su questo snapshot, mantenendo la coerenza e l'isolamento da modifiche apportate da altre transazioni o operazioni esterne che avvengono dopo la creazione dello snapshot;

4. le modifiche dalla transazione sono applicate a livello di transazione e sono visibili solo all'interno della stessa, quindi non influenzano la visibilità esterna fino alla conclusione della transazione.

2.4 Letture e Durability

È possibile che operazioni di lettura vedano i risultati delle scritture prima che siano rese durature, indipendentemente dal read concern o dalla configurazione del journaling.
In altri sistemi, si fa riferimento a questo concetto come *read uncommitted*.
Si possono verificare le due situazioni di seguito descritte.
Un'operazione di lettura può vedere il risultato dell'operazione di scrittura prima che di quest'ultima venga fatto l'acknowledge dall'applicazione client.
Una seconda situazione che può verificarsi è che le operazioni di lettura possono accedere a dati che potrebbero essere successivamente annullati in casi come il failover del set di replica. Tuttavia, questo non implica che le operazioni di lettura possano vedere documenti in uno stato parzialmente scritto o comunque incoerente.

2.5 Livelli di isolamento

In presenza di operazioni di lettura e scrittura concorrenti, MongoDB:

- Garantisce che le operazioni di lettura e scrittura siano atomiche rispetto al singolo documento. In altre parole, non sarà mai possibile accedere a un documento parzialmente aggiornato. Le operazioni di lettura e scrittura riferite ad un singolo documento sono serializzabili.
- Assicura la correttezza rispetto ai predicati delle query, per esempio eseguendo una *find* su una collection si avranno come risultato solo quei documenti che effettivamente rispettano il predicato indicato nella find. Lo stesso vale per l'aggiornamento.
- Dà garanzia di correttezza dell'ordinamento: un'operazione di lettura ordinata dei dati non viene inficiata da operazioni di scrittura concorrenti.

Quindi, in MongoDB, un'operazione di scrittura su un singolo documento è atomica, anche se questa modifica documenti annidati all'interno del singolo documento.

Questo non è vero per le operazioni di lettura e scrittura che accedono a più documenti contemporaneamente, che quindi non avvengono in modo transazionale.

Capitolo 3

Replicazione

La replicazione si riferisce alla pratica di mantenere copie multiple degli stessi dati su diversi nodi all'interno di un sistema. Con la replicazione, MongoDB è in grado di garantire alta disponibilità, ridondanza e l'affidabilità dei dati, anche in presenza di guasti o malfunzionamenti dei componenti del sistema poiché avere copie multiple degli stessi dati su diversi server permette di aumentare la fault tolerance.
Inoltre, se i client hanno bisogno di accedere agli stessi dati, le diverse operazioni di lettura possono essere eseguite sui diversi server. In questo senso, la replicazione può fornire una maggiore capacità di lettura, in quanto i client possono inviare operazioni di lettura a server diversi.

3.1 Replica Set

La replicazione in MongoDB è gestita tramite il concetto di Replica Set. Un Replica Set in MongoDB è un gruppo di istanze di *mongod* che forniscono ridondanza e alta disponibilità. Nell'architettura di un Replica set, ci sono diversi tipi di nodi:

- uno e un solo nodo primario,
- uno o più nodi secondari,
- un nodo arbitro (opzionale).

Il nodo primario è il nodo master che riceve tutte le operazioni di scrittura ed è l'unico a poter gestire tali operazioni.
I nodi secondari replicano le operazioni di scrittura dal primario per mantenere un insieme di dati identico.
Un nodo arbitro è un nodo che non contiene dati (non fornisce ridondanza di dati), ma ha lo scopo di garantire che sia sempre possibile raggiungere un quorum, ovvero il numero minimo di nodi attivi necessari per prendere decisioni, partecipando alle elezioni.
In totale quindi ci possono essere N (+1) nodi: un nodo primario, N-1 nodi secondari, eventualmente un nodo arbitro.
La configurazione minima consigliata per un Replica Set è data da tre membri: un nodo primario e due nodi secondari.

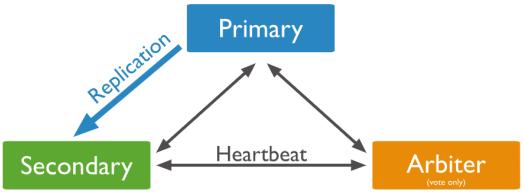

Figura 3.1: Struttura di un Replica Set

3.1.1 Nodo Primario

Il nodo primario accetta tutte le operazioni di scrittura e registra tutte le modifiche ai dati nel suo registro delle operazioni, cioè l'oplog. I secondari mantengono allineate le repliche, applicando quanto riportato nell'oplog ai loro dati, in modo che riflettano l'insieme di dati del primario.
Tutti i membri del Replica Set possono accettare operazioni di lettura ma, per impostazione predefinita, queste sono indirizzate al nodo primario, che è l'unico a fornire garanzie sull'aggiornamento dei dati letti. Come già discusso, per modificare tale comportamento predefinito in lettura è possibile specificare una preferenza di lettura attraverso la proprietà nota come *ReadPreference*. In questo modo un client può inviare le operazioni di lettura ai membri secondari.

Il replica set può avere al massimo un membro primario. Se il membro primario corrente diventa indisponibile, un'elezione determina il nuovo membro primario.

3.1.2 Nodo Secondario

Un nodo secondario mantiene una copia del dataset che è sul primario. Per applicare la replicazione dei dati, il secondario legge le operazioni dall'oplog del primario e le applica al suo dataset con una modalità asincrona.
Mentre il nodo primario di un replica set è uno soltanto, i nodi secondari possono essere anche più di uno.
Se le operazioni di scrittura sono ammesse solo al nodo primario, quelle di lettura vengono eseguite anche sui secondari, in base alla preferenza di lettura specificata.

3.1.3 Nodo Arbitro

Un arbitro è molto utile in alcune situazioni, come quando, avendo un primary e un secondary, i costi non permettono di aggiungere un altro nodo secondario. Il nodo arbitro partecipa all'elezione, ma non può esser eletto come primario non contiene una copia dei dati.
Esso dispone di un voto.
Per garantire la persistenza di una scrittura dopo il guasto di un nodo primario, il write concern *majority* richiede la conferma di un'operazione di scrittura da parte

della maggioranza dei nodi. Gli arbitri non memorizzano dati, ma contribuiscono al numero di nodi di un set di replica.
Tra gli arbitri e gli altri membri del replica set vengono scambiate alcune informazioni, quali i voti durante le elezioni, gli heartbeat e i dati di configurazione.

3.2 Processo di elezione

Per determinare quale membro del Replica Set è il nodo primario si utilizza il concetto di elezione. Un'elezione può essere avviata in specifiche circostanze, quali:

- avvio del Replica Set,
- aggiunta di un nuovo nodo al Replica Set,
- a seguito di interventi di manutenzione ($rs.stepDown(), rs.reconfig()$),
- perdita della connessione con il primario per un periodo di tempo superiore al timeout configurato, per impostazione predefinita pari a 10 secondi; i membri dei set si inviano reciprocamente heartbit (ping) ogni due secondi, ma se un heartbeat non viene restituito entro 10 secondi, gli altri membri contrassegnano il membro come inaccessibile.

In tali circostanze uno dei nodi richiede un'elezione per selezionare un nuovo nodo primario e riprendere automaticamente le normali operazioni.
Ogni nodo ha una priorità che influisce sia sulla tempistica che sull'esito delle elezioni: i secondari con priorità più alta convocano le elezioni relativamente prima dei secondari con priorità più bassa e hanno anche maggiori probabilità di vincere.
I membri con un valore di priorità pari a 0 non possono essere eletti e quindi non possono diventare nodi primari; tra questi è sempre presente il nodo arbitro che per definizione non può essere eletto.

Il Replica Set non può elaborare operazioni di scrittura finché l'elezione non è completata con successo. Al contrario, può continuare a servire query di lettura se tali query sono configurate per essere eseguite sui nodi secondari.

3.3 Replica Set Oplog

L'operations log (registro delle operazioni), abbreviato in oplog, è una speciale collezione che tiene traccia di tutte le operazioni che modificano i dati salvati nel database. Se un'operazione di scrittura fallisce o non modifica dati, non viene registrata nell'oplog.
Nella pratica, MongoDB applica le operazioni sul nodo primario e poi registra tali operazioni nell'oplog del nodo stesso. I membri secondari quindi copiano e applicano queste operazioni in un processo asincrono. Tutti i membri del set di

replica contengono una copia dell'oplog, che consente loro di mantenere lo stato corrente del database. Ogni membro secondario può importare le voci dell'oplog da qualsiasi altro membro.

3.4 Sincronizzazione dei dati

Per mantenere copie aggiornate all'interno Replica Set, i membri secondari sincronizzano o replicano i dati dagli altri membri. MongoDB utilizza due forme di sincronizzazione dei dati: la sincronizzazione iniziale e la replica. La sincronizzazione iniziale avviene per la prima volta quando MongoDB crea o ripristina un membro. La replicazione è la forma di sincronizzazione che avviene continuamente per mantenere ogni membro aggiornato con le modifiche ai dati del set di replica.

3.5 Scritture e letture in un Replica Set

Per un'applicazione client, il fatto che un'istanza di MongoDB sia in esecuzione come singolo server (standalone) o come insieme di repliche è trasparente. Tuttavia, MongoDB fornisce la possibilità di specificare differenti configurazioni per le operazioni di lettura e di scrittura all'interno di un Replica Set.

L'opzione *writeConcern* descrive il numero di membri portatori di dati, cioè i primari e i secondari, ma non gli arbitri, che devono fornire l'approvazione per un'operazione di scrittura prima che tale operazione venga considerata riuscita. Un membro può riconoscere un'operazione di scrittura solo dopo averla ricevuta e applicata con successo.
Possibili configurazioni sono:

- w : "*majority*" è il criterio di scrittura predefinito che richiede che l'operazione di scrittura siano state confermate, in modo duraturo, dalla maggioranza dei membri votanti (escluso il nodo arbitro);
- $w : 1$ è il criterio di scrittura che richiede la sola conferma da parte del nodo primario del set;
- $w : n$, dove n è un valore numerico superiore a 1, è il criterio di scrittura che richiede la conferma da parte del nodo primario e di un numero di nodi secondari necessario per raggiungere il valore specificato.

Per le operazioni di scrittura con $w > 1$ o w : "*majority*", il nodo primario deve attendere che un numero sufficiente di nodi secondari riconosca la scrittura prima di restituire il *write concern acknowledgment* dell'operazione di scrittura. Al contrario, nel caso di $w : 1$, il nodo primario può restituire il *write concern acknowledgment* dell'operazione di scrittura non appena applica localmente la stessa.

Aumentare il numero di membri a cui è richiesto l'acknowledgment di una scrittura permette di ridurre la possibilità di un roll back dei dati scritti. Tuttavia, specificare un alto livello di write concern può aumentare la latenza, poiché il client deve attendere di ricevere il *write concern acknowledgment* richiesto. È necessario un trade-off tra gli obiettivi di prestazione e i requisiti di durabilità dei dati dell'applicazione.

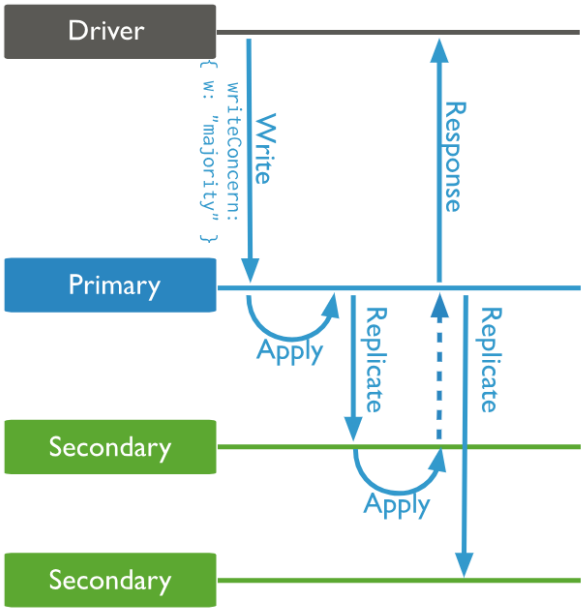

Figura 3.2: Meccanismo di replicazione

Capitolo 4

Sperimentazione

Nel presente capitolo ci concentreremo sulla sperimentazione della gestione delle transazioni in MongoDB, esplorando le capacità e le limitazioni di questo database NoSQL.
Le transazioni multi-documento, introdotte con MongoDB 4.0, offrono una maggiore affidabilità e integrità dei dati, rendendo MongoDB una scelta valida anche per applicazioni critiche che richiedono un'elevata coerenza.

Il codice scritto per la sperimentazione è consultabile alla repository GitHub.

Come primo passo, è stato costruito un ambiente appropriato su MongoDB Atlas. È stato creato e configurato un cluster, nella versione gratuita.
Il cluster gratuito è noto come M0, ed è il livello base nel modello di pricing di MongoDB Atlas; questo presenta le seguenti caratteristiche e limitazioni:

- un Replica Set costituito da un nodo primario e due secondari;
- il cluster può essere distribuito su AWS, Google Cloud Platform (GCP) o Microsoft Azure;
- possibilità limitata di scelta della regione, il che influisce sui requisiti di localizzazione geografica e sulla latenza;
- limitazione delle capacità: 512 MB di spazio di archiviazione e di RAM;
- limitazione delle funzionalità: non si hanno i privilegi per modificare alcuni parametri, come il tempo di ritardo per la propagazione delle modifiche sui nodi secondari.

Si è scelto di utilizzare il driver MongoDB per il linguaggio di programmazione Python.
Si è anche installata l'ultima versione disponibile della shell di MongoDB.

Ciascuna prova, infatti, è stata eseguita in un primo momento sulla shell e poi utilizzando il linguaggio di programmazione Python e PyMongo, la libreria ufficiale di MongoDB per Python.

Nelle prossime pagine saranno riportati solo i risultati ottenuti con PyMongo, dato che il comportamento sulla mongoshell è pressoché lo stesso.
L'unica differenza riscontrata riguarda l'uso del valore di write concern *w:0*, che è permesso sulla mongo shell, ma, che se usato in PyMongo, genera l'errore:
transactions do not support unacknowledged write concern: WriteConcern(w=0).

Ai fini della sperimentazione, si sono creati un database, delle collezioni e dei documenti di esempio.

Per implementare le transazioni in MongoDB, indipendentemente dal fatto che vengano eseguite sulla shell o usando PyMongo, è necessario seguire i seguenti passi:

- inizializzare una sessione;
- iniziare la transazione;
- eseguire operazioni di lettura e scrittura;
- concludere la transazione mandandola in commit o in abort;
- chiudere la sessione.

Il capitolo, dopo una prima sezione che introduce PyMongo e una seconda che descrive brevemente il dataset, mostra il comportamento di MongoDB in presenza di anomalie quali la perdita di aggiornamento, la lettura sporca, l'inserimento fantasma, l'aggiornamento fantasma e la write skew.
Si precisa che il codice di esempio mostrato riporta solo una combinazione di read e write concern per ciascun tentativo. Tutte le altre combinazioni sono state testate e non hanno determinato differenze nei risultati. Per semplicità e per evitare ridondanze, il codice non viene ripetuto per ciascun tentativo con le diverse combinazioni. Basterà modificare i parametri direttamente nel file in base alle proprie esigenze.

4.1 Database

Il database utilizzato, *negozio_abbigliamento*, archivia informazioni su articoli di abbigliamento disponibili in un negozio e sugli acquisti effettuati.
Esso si compone di due collezioni principali:

1. collezione *capi_abbigliamento*, che contiene i dati relativi agli articoli di abbigliamento disponibili nel negozio,
2. collezione *scontrini*, che registra gli acquisti avvenute, inclusi i dettagli degli articoli acquistati e il totale complessivo.

La collezione *capi_abbigliamento* si compone di documenti che rappresentano i capi di abbigliamento. Ogni documento rappresenta un capo di abbigliamento disponibile nel negozio e contiene i seguenti campi:

- *_id*, un identificatore univoco per ogni articolo, generato automaticamente da MongoDB (ObjectId);
- *capoID*, un identificatore univoco per ogni articolo gestito tramite autoincremento;
- *nome*, il nome dell'articolo (stringa);
- *prezzo*, il prezzo dell'articolo (numero decimale);
- *colore*, il colore dell'articolo (stringa);
- *disponibilita*, un campo valorizzato da un documento nidificato che rappresenta la disponibilità dell'articolo nelle diverse taglie; il documento innestato consiste in un mapping dove la chiave è la taglia (ad esempio, "S", "M", "L", "XL") e il valore è il numero di pezzi disponibili per quella taglia (intero).

La collezione *scontrini* si compone di documenti che rappresentano gli acquisti effettuati nel negozio, inclusi i dettagli degli articoli acquistati e il totale complessivo della transazione. Ogni documento rappresenta uno scontrino e contiene i seguenti campi:

- *_id*, un identificatore univoco per ogni scontrino, generato automaticamente da MongoDB (ObjectId);
- *scontrinoID*, un identificatore univoco per ogni scontrino gestito tramite autoincremento;
- *data*, la data in cui è avvenuto l'acquisto (datetime);
- *articoli*, la lista degli articoli acquistati, in cui ogni voce è un documento che riporta i dettagli dell'articolo stesso;
- *totale_complessivo*, il prezzo totale degli acquisti inclusi nello scontrino (numero decimale).

4.2 PyMongo

PyMongo è una distribuzione Python che mette a disposizione degli strumenti per lavorare con MongoDB.
Per poter utilizzare la libreria PyMongo, è necessario installarla attraverso il comando *pip* riportato di seguito.

```
python -m pip install pymongo
python -m pip install --upgrade pymongo
```

Listing 4.1: Installazione e aggiornamento di PyMongo.

Per potersi connettere a un cluster di MongoDB Atlas basta passare all'istanza di *MongoClient* la stringa di connessione disponibile nella sezione *Database*, premendo sulla stringa *Connect* accanto al nome del cluster e selezionando il Driver Python.

```
from pymongo import MongoClient

client = MongoClient("mongodb+srv://username:password@cluster1.1mnlttb.mongodb.net/?appName=
    mongosh+2.2.10")
```

Listing 4.2: Esempio di connessione a un cluster MongoDB Atlas.

Per fare il testing sul database creato da noi al codice presente su GitHub è necessario sostituire alla stringa "..." la stringa di connessione del proprio cluster.
Per poter accedere ai database e alle collezioni presenti sul cluster si possono usare due sintassi diverse, la *attribute style* o la *dictionary style*.

```

db = client.negozio_abbigliamento
capi_collection = db.capi_abbigliamento
```

Listing 4.3: Accesso attribute style a un database e a una collezione.

```

db = client["negozio_abbigliamento"]
capi_collection = db["capi_abbigliamento"]
```

Listing 4.4: Accesso dictionary style a un database e a una collezione.

4.2.1 Ricerca dei documenti

A questo punto è possibile recuperare un documento appendendo all'oggetto collezione la funzione *find_one()*. È possibile passarle un filtro, ovvero un documento che rappresenta quali caratteristiche si vuole che abbia il documento che stiamo cercando. Viene restituito il primo documento che soddisfa il filtro. Se non c'è nessuna corrispondenza, viene restituito *None*.
Se non si specifica nessun filtro, la funzione restituisce il primo documento della collezione.

Per poter recuperare più documenti, PyMongo mette a disposizione la funzione *find()*. Se non si specifica nessun filtro, essa restituisce tutti i documenti presenti nella collezione.
Essa restituisce un'istanza della classe *Cursor*, utile per iterare sui risultati delle query Mongo.

Ad esempio, su un cursor è possibile applicare la funzione *sort()* che ordina il risultato della find secondo il campo specificato come primo parametro. Il secondo parametro permette di specificare se l'ordine desiderato è crescente o decrescente.

```
cursor = capi_collection.find()
cursor = cursor.sort('nome', pymongo.ASCENDING)

for doc in cursor:
    print(doc)
```

Listing 4.5: Ordinamento del risultato di una find.

4.2.2 Inserimento di documenti

Per l'inserimento di un documento è possibile servirsi della funzione *insert_one()* e per un inserimento bulk si può fare ricorso alla funzione *insert_many()*.
La funzione *insert_one()* prende in input un documento e restituisce un *result*, in particolare un oggetto della classe InsertOneResult. Un oggetto di questa classe contiene la proprietà *inserted_id*, ovvero l'id documento appena inserito, generato automaticamente da MongoDB se non viene specificato esplicitamente nel documento al momento dell'inserimento.

```
result = capi_abbigliamento.insert_one(nuovo_capo)

document_id = result.inserted_id
print(f"_id del documento inserito: {document_id}")
```

Listing 4.6: Inserimento di un documento.

Analogamente, la funzione *insert_many()* richiede una lista di documenti in input e restituisce un oggetto InsertManyResult, contenente *inserted_ids*, una lista degli id appena inseriti.

4.2.3 Aggiornamento di documenti

Per poter aggiornare un documento si può applicare la funzione *update_one()* alla collezione.
Necessita di due parametri obbligatori, un documento che permette di recuperare il documento da aggiornare e un documento che specifica le modifiche da apportare. Il discorso è analogo per *update_many()*.
Come le funzioni di inserimento, l'aggiornamento restituisce un *result*, in particolare un UpdateResult. Esso è caratterizzato dalle proprietà:

- *matched_count*, ovvero il numero di documenti che corrispondono al filtro specificato. 0 o 1 nel caso di *update_one()*;
- *modified_count*, che rappresenta il numero di documenti modificati,
- *raw_result*, che è un dizionario di dati grezzi contenente informazioni dettagliate sull'operazione di aggiornamento, utili per il debug o per ottenere informazioni più approfondite sull'esito dell'operazione;

- *upserted_id*, che rappresenta l'_id del documento inserito, se c'è stato un upsert, ovvero l'inserimento di un documento non presente.

4.2.4 Cancellazione di documenti

Per poter eliminare uno o più documenti esistono la funzione *delete_one()* e la *delete_many()*, rispettivamente.
Il parametro obbligatorio è un documento filtro, per trovare il documento da eliminare. Se non si specifica nessun filtro, la funzione elimina il primo documento presente nella collezione a cui è applicato.

Per poter eliminare più di un documento, esiste la funzione *delete_many()*, a cui si può passare il filtro per individuare i documenti da eliminare. Non specificarlo comporta l'eliminazione di tutti i documenti della collezione.

Anche le funzioni di eliminazione restituiscono un risultato, in particolare un oggetto della classe *DeleteResult*. Una delle proprietà di questa classe è *deleted_count*, che indica il numero di documenti eliminati.

4.2.5 Esempi di uso delle operazioni CRUD

In questa sezione viene riportato un codice che applica quanto discusso nelle sottosezioni precedenti.

```
import pprint

from bson import ObjectId, Decimal128
from pymongo import MongoClient

connection_string = "..."
client = MongoClient(connection_string)

db = client.negozio_abbigliamento

capi_abbigliamento = db.capi_abbigliamento
scontrini = db.scontrini

nuovo_capo = {
        "nome": "Collana",
        "prezzo": Decimal128("29.99"),
        "colore": "oro",
        "disponibilita": {"S": 2, "M": 5, "L": 2}
    }

result = capi_abbigliamento.insert_one(nuovo_capo)

document_id = result.inserted_id
print(f"_id del documento inserito: {document_id}")

document_to_find = {"_id": ObjectId(document_id)}
pprint.pprint(capi_abbigliamento.find_one(document_to_find))

document_to_find = {"_id": ObjectId(document_id)}

cursor = capi_abbigliamento.find(document_to_find)

num_docs = 0
for document in cursor:
    num_docs += 1
    pprint.pprint(document)
```

```
37      print()
38  print("Numero di documenti trovati: " + str(num_docs))
39
40  document_to_update = {"_id": ObjectId(document_id)}
41  pprint.pprint(capi_abbigliamento.find_one(document_to_update))
42
43  add_product = {"$inc": {"prezzo": Decimal128("10")}}
44  result = capi_abbigliamento.update_one(document_to_update, add_product)
45  print("Documenti aggiornati: " + str(result.modified_count))
46
47  pprint.pprint(capi_abbigliamento.find_one(document_to_update))
48
49  documents_to_delete = {"nome": "Collana"}
50
51  result = capi_abbigliamento.delete_many(documents_to_delete)
52  print("Documenti eliminati: " + str(result.deleted_count))
53
54
55  client.close()
```

Listing 4.7: Esempi di uso delle operazioni CRUD in PyMongo.

4.2.6 Transazioni con PyMongo

Gli esempi di transazione discussi nelle sezioni successive sono strutturati secondo lo schema di seguito descritto.

Innanzitutto, per poter creare una transazione è necessario avere una connessione attiva al database.

Successivamente, bisogna definire la callback che specifica l'insieme di operazioni che si desidera eseguire nella transazione.
Il parametro da inserire obbligatoriamente nella callback è quello che indica la sessione. Possono essere aggiunti ulteriori parametri, se necessari alle operazioni della transazione.
Il riferimento alle collezioni utili allo svolgimento delle transazioni di norma viene preso all'interno della callback.
Completate queste operazioni preliminari, si possono scrivere le operazioni della transazione. Si ricorda che è necessario passare la sessione a ogni operazione.

La sessione può essere iniziata usando il metodo *start_session* sull'oggetto client in uno statement *with*.
Il metodo della sessione *with_transaction* inizia la transazione, esegue la callback e poi esegue il commit o l'abort. La funzione di callback è il parametro obbligatorio di *with_transaction*; spesso, per poter passare argomenti addizionali, la callback viene racchiusa in un'ulteriore funzione, la *callback_wrapper*.
In sintesi, con l'istruzione
session.with_transaction(callback_wrapper) è possibile eseguire la transazione.

L'importanza nell'uso di una callback sta nel fatto che in tal modo il driver ritenta automaticamente la transazione quando incontra alcuni errori segnalati dal server.

```python
from datetime import datetime
from pymongo import MongoClient

connection_string = "..."
client = MongoClient(connection_string)

def callback(session, articolo=None, taglia=None):

    capi_abbigliamento = session.client.negozio_abbigliamento.capi_abbigliamento
    scontrini = session.client.negozio_abbigliamento.scontrini

    articolo = capi_abbigliamento.find_one({'nome': articolo}, session=session)
    prezzo_articolo = articolo.get("prezzo")

    campo_da_aggiornare = f"disponibilita.{taglia}"

    capi_abbigliamento.update_one(
        {"nome": "Felpa"},
        {"$inc": {campo_da_aggiornare: -1}},
        session=session,
    )

    scontrino_da_inserire = {
        "data": datetime.strptime(str(datetime.now().date()), "%Y-%m-%d"),
        "articoli": [
            {"nome": "Felpa", "quantita": 1, "prezzo_totale": prezzo_articolo, "taglia": "L"},
        ],
        "totale_complessivo": prezzo_articolo
    }

    scontrini.insert_one(
        scontrino_da_inserire,
        session=session,
    )

    print("Transaction successful")

    return

def callback_wrapper(s):
    callback(
        s,
        "Felpa",
        "L"
    )

with client.start_session() as session:
    session.with_transaction(callback_wrapper)


client.close()
```

Listing 4.8: Un esempio di transazione.

La transazione di esempio rappresenta l'acquisto di un capo di abbigliamento, operazione atomica che include l'inserimento di uno scontrino e il decremento della quantità dell'articolo acquistato.
La funzione di callback, oltre al parametro della sessione, accetta anche altri due parametri, il nome del capo da acquistare e la taglia.

4.3 Validazione dello schema

MongoDB mette a disposizione JSON Schema Validation, una funzionalità che consente di definire e applicare regole di convalida dei dati a livello di schema per i documenti memorizzati in una collezione. Tale funzionalità è utile per garantire che

i documenti inseriti o aggiornati in una collezione soddisfino determinati requisiti strutturali e di contenuto.

Nel nostro database di esempio, nella collezione *capi_abbigliamento*, all'atto della creazione, si sono aggiunti i seguenti vincoli:

- i campi *prezzo*, *nome* e *disponibilita* sono obbligatori,
- il campo *nome* deve essere una stringa;
- il campo *prezzo* deve essere un Decimal128, deve essere almeno 0 e massimo 10000;
- il campo *disponibilità* è un oggetto le cui chiavi siano le taglie *XS*, *S*, *M*, *L*, *XL*, che possono esser presenti o meno, ma le uniche stringhe a poter essere presenti sono queste.

Per la collection scontrini, all'atto della creazione, si sono definiti i seguenti vincoli:

- i campi *data*, *articoli* e *totale_complessivo* devono essere necessariamente presenti;
- *data* deve essere di tipo data;
- il campo *articoli* deve essere un array, con gli elementi *nome*, *quantita*, *taglia* e *prezzo_totale*.
  - *nome* deve essere una stringa;
  - *quantita* deve essere maggiore o uguale a 1;
  - *taglia* deve essere una strina tra *XS, S, M, L, XL*;
  - *prezzo_totale* deve essere un Decimal128 con valore minimo 0,
  - *totale_complessivo* deve essere un Decimal128 con valore minimo 0, valore che è la somma dei prezzi totali degli articoli.

Essendo MongoDB un database schema-less, non esiste il concetto di vincolo come inteso nei DBMS relazionali. Il campo *totale_complessivo* dello scontrino rappresenta la somma del prezzo degli articoli, quindi tra questi campi dovrebbe esistere un vincolo, che però, in quanto più articolato rispetto ai vincoli precedentemente presentati, deve essere espresso e gestito a livello applicativo.

```
1  from pymongo import MongoClient
2
3  connection_string = "..."
4  client = MongoClient(connection_string)
5
6  db = client['negozio_abbigliamento']
7
8  validator = {
9      "$jsonSchema": {
10         "bsonType": "object",
11         "title": "Validazione dei documenti della collezione capi_abbigliamento",
```

```
        "required": ["prezzo", "nome", "disponibilita"],
        "properties": {
            "nome": {
                "bsonType": "string",
                "description": "nome deve essere una stringa ed e' obbligatorio"
            },
            "prezzo": {
                "bsonType": "decimal",
                "minimum": 0,
                "maximum": 10000,
                "description": "prezzo deve essere un Decimal128 positivo"
            },
            "disponibilita": {
                "bsonType": "object",
                "properties": {
                    "XS": {
                        "bsonType": "int",
                        "minimum": 0,
                        "description": "XS deve essere un intero maggiore di 0"
                    },
                    "S": {
                        "bsonType": "int",
                        "minimum": 0,
                        "description": "S deve essere un intero maggiore di 0"
                    },
                    "M": {
                        "bsonType": "int",
                        "minimum": 0,
                        "description": "M deve essere un intero maggiore di 0"
                    },
                    "L": {
                        "bsonType": "int",
                        "minimum": 0,
                        "description": "L deve essere un intero maggiore di 0"
                    },
                    "XL": {
                        "bsonType": "int",
                        "minimum": 0,
                        "description": "XL deve essere un intero maggiore di 0"
                    }
                },
                "patternProperties": {
                    "^(?!XS|S|M|L|XL$).*": {
                        "bsonType": "null",
                        "description": "Nessun'altra chiave e' permessa in disponibilita"
                    }
                },
                "description": "disponibilita deve essere un oggetto con chiavi 'XS',"
                               " 'S', 'M', 'L', 'XL' e con valori interi positivi"
            }
        }
    }
}

db.create_collection(
    'capi_abbigliamento',
    validator=validator
)

print("La collezione capi_abbigliamento è stata correttamente creata con lo schema di
     validazione.")

validator_scontrini = {
    "$jsonSchema": {
        "bsonType": "object",
        "title": "Validazione dei documenti inseriti nella collection scontrini",
        "required": ["data", "articoli", "totale_complessivo"],
        "properties": {
            "data": {
                "bsonType": "date",
                "description": "data deve essere una data ed e' obbligatorio"
            },
            "articoli": {
                "bsonType": "array",
                "items": {
```

```
                    "bsonType": "object",
                    "required": ["nome", "quantita", "taglia", "prezzo_totale"],
                    "properties": {
                        "nome": {
                            "bsonType": "string",
                            "description": "nome deve essere una stringa ed e' obbligatorio"
                        },
                        "quantita": {
                            "bsonType": "int",
                            "minimum": 1,
                            "description":"quantita deve essere un intero maggiore o uguale a 1"
                        },
                        "taglia": {
                            "bsonType": "string",
                            "enum": ["XS", "S", "M", "L", "XL"],
                            "description": "taglia deve essere una delle seguenti: 'XS', 'S', 'M
    ',"
                                           " 'L', 'XL'"
                        },
                        "prezzo_totale": {
                            "bsonType": "decimal",
                            "minimum": 0,
                            "description": "prezzo_totale deve essere un numero positivo"
                        }
                    }
                },
                "description": "articoli deve essere un array di oggetti articoli"
            },
            "totale_complessivo": {
                "bsonType": "decimal",
                "minimum": 0,
                "description": "totale_complessivo deve essere un numero positivo e rappresenta"
                               " la somma dei prezzi totali degli articoli"
            }
        }
    }
}

db.create_collection(
    'scontrini',
    validator=validator_scontrini
)

print("La collezione scontrini è stata correttamente creata con lo schema di validazione.")
```

Listing 4.9: Definizione degli schemi di validazione

4.4 Definizione dei Trigger

I trigger sono blocchi di codice che vengono eseguiti automaticamente al verificarsi di determinati eventi nel database.
I trigger si basano su change stream di MongoDB per osservare le modifiche in tempo reale in una raccolta. Un change stream è una serie di eventi che descrivono ciascuno un'operazione su un documento della collezione.
Un'applicazione definisce un singolo change stream per ogni raccolta a cui si riferisce almeno un trigger abilitato; se per una raccolta sono abilitati più trigger, tutti condividono lo stesso change stream. I change stream sfruttano il log delle operazioni (oplog) di MongoDB per rilevare i cambiamenti.

Si può configurare e attivare un trigger su MongoDB Atlas, facendo clic sulla voce "Triggers" nel menu di navigazione laterale e facendo clic su "Add a Trigger".

A questo punto, dalla schermata che si aprirà, è possibile configurare opportunamente il trigger e la funzione da eseguire in risposta al rilevamento dell'evento di interesse.

Nel nostro caso si sono definiti due trigger, uno per la collezione *capi_abbigliamento* e uno per la collezione *scontrini*, per gestire, all'inserimento di un nuovo documento, l'auto-incremento degli identificatori *capoId* e *scontrinoId*.
Per impostazione predefinita MongoDB genera automaticamente un identificatore *_id* come *ObjectId*, un tipo di dati binari a 12 byte. Sebbene questo garantisca l'unicità, può essere difficile da leggere e manipolare, per questo, in alcune applicazioni, è utile avere un ulteriore identificatore. In questo caso si è deciso di specificare un identificatore come un intero auto-incrementale, gestito tramite trigger.
Lo scopo dei seguenti trigger è appunto quello di creare un identificatore unico e auto-incrementale per i documenti delle due collezioni, mantenendo *_id* come identificatore primario, autogestito da MongoDB.

Di seguito sono riportate le funzioni dei due trigger; il loro comportamento è analogo.

Si dichiara una funzione asincrona che accetta un parametro *changeEvent*, il quale rappresenta l'evento che ha innescato il trigger. Questo evento contiene informazioni dettagliate sul cambiamento avvenuto nel database, in questo caso l'inserimento di un nuovo documento nella collezione in riferimento alla quale è definito il trigger.
changeEvent.ns.db contiene il nome del database in cui è avvenuto il cambiamento; *ns* sta per namespace ed è un oggetto che contiene sia il nome del database sia il nome della collezione. *collection*(*changeEvent.ns.coll*) accede dinamicamente alla collezione in cui è avvenuto il cambiamento. *collection*("*counters*") accede dinamicamente ad una collezione specifica, definita proprio per gestire l'auto-incremento. *counters* si compone di una serie di documenti, tanti quante sono le collezioni del database, ognuno con *_id* pari al namespace della collezione a cui si riferisce, così da assicurare che ogni collezione abbia il proprio contatore unico e separato; tra i campi di tali documenti è inoltre specificato un contatore incrementale *seq_value*, che permette di tenere traccia dell'ultimo valore utilizzato come identificatore per ogni collezione.
Con un'operazione atomica si individua e si aggiorna il documento associato alla collezione modificata, incrementando di una unità il campo *seq_value*. Se il documento non esiste, cioè se non è stato ancora creato un contatore per quella collezione, questo viene creato automaticamente con *_id* impostato su *changeEvent.ns* e *seq_value* inizializzato a 1. Infine, viene aggiunto un nuovo campo (*capoId*/*scontrinoId*) nel documento appena inserito con il valore incrementale *seq_value* ottenuto dalla collezione *counters*.

```
1 exports = async function(changeEvent) {
2     var docId = changeEvent.fullDocument._id;
```

```
    const countercollection = context.services.get("Cluster1").db(changeEvent.ns.db).collection
     ("counters");
    const capicollection = context.services.get("Cluster1").db(changeEvent.ns.db).collection(
     changeEvent.ns.coll);
    var counter = await countercollection.findOneAndUpdate({_id: changeEvent.ns },{ $inc: {
     seq_value: 1 }}, { returnNewDocument: true, upsert : true});
    var updateRes = await capicollection.updateOne({_id : docId},{ $set : {capoId : counter.
     seq_value}});
    console.log(`Updated ${JSON.stringify(changeEvent.ns)} with counter ${counter.seq_value}
     result : ${JSON.stringify(updateRes)}`);
    };
```

Listing 4.10: Trigger per la collezione *capi_abbigliamento*

```
exports = async function(changeEvent) {
    var docId = changeEvent.fullDocument._id;
    const countercollection = context.services.get("Cluster1").db(changeEvent.ns.db).collection
     ("counters");
    const scontrinicollection = context.services.get("Cluster1").db(changeEvent.ns.db).
     collection(changeEvent.ns.coll);
    var counter = await countercollection.findOneAndUpdate({_id: changeEvent.ns },{ $inc: {
     seq_value: 1 }}, { returnNewDocument: true, upsert : true});
    var updateRes = await scontrinicollection.updateOne({_id : docId},{ $set : {scontrinoId :
     counter.seq_value}});
    console.log(`Updated ${JSON.stringify(changeEvent.ns)} with counter ${counter.seq_value}
     result : ${JSON.stringify(updateRes)}`);
    };
```

Listing 4.11: Trigger per la collezione *scontrini*

4.5 Popolamento del database

Prima di procedere con la sperimentazione è necessario inserire alcuni documenti all'interno delle collezioni che costituiscono il database.
Lo script *popolamento_db.py* è un file che contiene il codice per inserire dati iniziali nel database.
Le ultime righe di codice, quelle commentate, sono un esempio di documenti che violano lo schema di validazione e che quindi determinerebbero un'eccezione qualora si tentasse di inserirli.

```
from datetime import datetime
from bson import Decimal128
from pymongo import MongoClient

connection_string = "..."
client = MongoClient(connection_string)

db = client['negozio_abbigliamento']

counters = db['counters']
counters.delete_many({})

collezione = db['capi_abbigliamento']
collezione.delete_many({})

documenti = [
    {
        "nome": "Maglietta",
        "prezzo": Decimal128("19.99"),
        "colore": "rosso",
        "disponibilita": {"S": 30, "M": 50, "L": 20}
    },
    {
```

```
        "nome": "Jeans",
        "prezzo": Decimal128("49.99"),
        "colore": "blu",
        "disponibilita": {"S": 25, "M": 60, "L": 40, "XL": 15}
    },
    {
        "nome": "Felpa",
        "prezzo": Decimal128("39.99"),
        "colore": "grigio",
        "disponibilita": {"M": 25, "L": 15, "XL": 30}
    },
    {
        "nome": "Giacca",
        "prezzo": Decimal128("74.99"),
        "colore": "beige",
        "disponibilita": {"S": 10, "M": 15, "L": 8}
    },
    {
        "nome": "Gonna",
        "prezzo": Decimal128("29.99"),
        "colore": "verde",
        "disponibilita": {"S": 40, "M": 30}
    },
    {
        "nome": "Pantaloni corti",
        "prezzo": Decimal128("24.99"),
        "colore": "blu",
        "disponibilita": {"S": 20, "M": 60, "L": 30}
    },
    {
        "nome": "Pantaloni",
        "prezzo": Decimal128("49.99"),
        "colore": "beige",
        "disponibilita": {"S": 10, "M": 60, "L": 20}
    },
    {
        "nome": "Camicia",
        "prezzo": Decimal128("34.99"),
        "colore": "bianco",
        "disponibilita": {"S": 20, "M": 15, "L": 25, "XL": 10}
    },
    {
        "nome": "Abito", "prezzo": Decimal128("99.99"), "colore": "beige",
        "disponibilita": {"S": 15, "M": 10, "L": 5}
    },
    {
        "nome": "Cappotto",
        "prezzo": Decimal128("129.99"),
        "colore": "marrone",
        "disponibilita": {"M": 8, "L": 5, "XL": 10}
    },
    {
        "nome": "Maglione",
        "prezzo": Decimal128("59.99"),
        "colore": "blu",
        "disponibilita": {"S": 10, "M": 35, "L": 20, "XL": 15}
    }
]

collezione.insert_many(documents=documenti)

print("Documenti inseriti con successo")


collezione_scontrini = db['scontrini']
collezione_scontrini.delete_many({})

scontrini = [
    {
        "data": datetime.strptime("2024-07-01", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Maglietta", "quantita": 2, "prezzo_totale":
                Decimal128("39.98"), "taglia": "M"},
            {"nome": "Jeans", "quantita": 1, "prezzo_totale":
                Decimal128("49.99"), "taglia": "XS"}
```

```
        ],
        "totale_complessivo": Decimal128("89.97")
    },
    {
        "data": datetime.strptime("2024-07-02", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Felpa", "quantita": 1, "prezzo_totale":
                Decimal128("39.99"), "taglia": "L"},
            {"nome": "Cappotto", "quantita": 1, "prezzo_totale":
                Decimal128("129.99"), "taglia": "XS"}
        ],
        "totale_complessivo": Decimal128("169.98")
    },
    {
        "data": datetime.strptime("2024-07-03", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Gonna", "quantita": 3, "prezzo_totale":
                Decimal128("89.97"), "taglia": "XS"},
            {"nome": "Maglietta", "quantita": 1, "prezzo_totale":
                Decimal128("19.99"), "taglia": "XL"}
        ],
        "totale_complessivo": Decimal128("109.96")
    },
    {
        "data": datetime.strptime("2024-07-04", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Camicia", "quantita": 2, "prezzo_totale":
                Decimal128("69.98"), "taglia": "XS"},
            {"nome": "Pantaloni corti", "quantita": 1, "prezzo_totale":
                Decimal128("24.99"), "taglia": "M"}
        ],
        "totale_complessivo": Decimal128("94.97")
    },
    {
        "data": datetime.strptime("2024-07-05", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Abito", "quantita": 1, "prezzo_totale":
                Decimal128("99.99"), "taglia": "M"},
            {"nome": "Maglione", "quantita": 1, "prezzo_totale":
                Decimal128("59.99"), "taglia": "M"}
        ],
        "totale_complessivo": Decimal128("159.98")
    },
    {
        "data": datetime.strptime("2024-07-06", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Giacca", "quantita": 1, "prezzo_totale":
                Decimal128("89.99"), "taglia": "S"},
            {"nome": "Jeans", "quantita": 1, "prezzo_totale":
                Decimal128("49.99"), "taglia": "XS"}
        ],
        "totale_complessivo": Decimal128("139.98")
    },
    {
        "data": datetime.strptime("2024-07-07", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Felpa", "quantita": 2, "prezzo_totale":
                Decimal128("79.98"), "taglia": "XS"},
            {"nome": "Maglietta", "quantita": 3, "prezzo_totale":
                Decimal128("59.97"), "taglia": "XS"}
        ],
        "totale_complessivo": Decimal128("139.95")
    },
    {
        "data": datetime.strptime("2024-07-08", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Cappotto", "quantita": 1, "prezzo_totale":
                Decimal128("129.99"), "taglia": "XL"},
            {"nome": "Gonna", "quantita": 2, "prezzo_totale":
                Decimal128("59.98"), "taglia": "XL"}
        ],
        "totale_complessivo": Decimal128("189.97")
    },
    {
        "data": datetime.strptime("2024-07-09", "%Y-%m-%d"),
```

```
        "articoli": [
            {"nome": "Abito", "quantita": 2, "prezzo_totale":
                Decimal128("199.98"), "taglia": "XS"},
            {"nome": "Camicia", "quantita": 1, "prezzo_totale":
                Decimal128("34.99"), "taglia": "M"}
        ],
        "totale_complessivo": Decimal128("234.97")
    },
    {
        "data": datetime.strptime("2024-07-10", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Maglione", "quantita": 1, "prezzo_totale":
                Decimal128("59.99"), "taglia": "XS"},
            {"nome": "Pantaloni corti", "quantita": 2, "prezzo_totale":
                Decimal128("49.98"), "taglia": "XL"}
        ],
        "totale_complessivo": Decimal128("109.97")
    }
]

collezione_scontrini.insert_many(documents=scontrini)


print("Scontrini inseriti con successo")

'''
db.capi_abbigliamento.insert_one({
    "nome": "Maglietta",
    "prezzo": Decimal128("19.99"),
    "colore": "rosso",
    "disponibilita": {"S": 30, "M": 50, "L": 20, "K":2}
})
'''
'''
db.capi_abbigliamento.insert_one({
        "prezzo": Decimal128("19.99"),
        "colore": "rosso",
        "disponibilita": {"S": 30, "M": 50, "L": 20, "XL": 2}
})
'''
'''
db.scontrini.insert_one({
        "data": datetime.strptime("2024-07-10", "%Y-%m-%d"),
        "articoli": [
            {"nome": "Felpa", "quantita": -1, "prezzo_totale": Decimal128("39.99"), "taglia":"XL
    "},
            {"nome": "Cappotto", "quantita": 1, "prezzo_totale": Decimal128("129.99"), "taglia
    ":"XL"}
        ],
        "totale_complessivo": Decimal128("69.98")
})
'''
```

Listing 4.12: Popolazione del database

4.6 Considerazioni sulla replicazione

Come già accennato nei capitoli precedenti, la replicazione fornisce ridondanza e aumenta la disponibilità dei dati. Le copie multiple dei dati su server di database diversi garantiscono maggiore tolleranza ai guasti. Quando si parla di tempo di replicazione ci si riferisce al tempo necessario perché i dati si propaghino da un nodo primario a uno o più nodi secondari all'interno di un replica set.

Avere un basso tempo di replicazione aumenta la probabilità che i dati siano sincronizzati tra il nodo primario e i nodi secondari, riducendo la possibilità di leggere dati

obsoleti. Infatti, se il nodo primario ha problemi, uno dei nodi secondari diventa il nuovo nodo primario e il nuovo nodo primario è il più aggiornato possibile. Inoltre, se il set di replica è configurato per leggere dai nodi secondari, un tempo di replicazione minore comporta le letture più coerenti.

Di default, MongoDB replica le modifiche ai nodi secondari nel minor tempo possibile. È possibile introdurre un ritardo andando a modificare il parametro secondaryDelaySecs nella configurazione dei membri del replica set, che di default è impostato a 0. Tale parametro rappresenta il ritardo, in secondi, che si vuole imporre per la propagazione delle modifiche dal primario a un certo membro secondario del replica set.

La documentazione mostra un esempio di modifica del tempo di replicazione di un certo nodo, riportato nel codice di seguito.

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].secondaryDelaySecs = 3600
rs.reconfig(cfg)
```

Listing 4.13: Modifica del tempo di replicazione

Nel nostro caso, nel momento in cui si esegue la prima riga, viene restituito un errore, che avvisa di non essere autorizzati a compiere quell'operazione.
Essendo MongoDB Atlas un servizio di database gestito, mette a disposizione un livello di astrazione e di controllo sulla configurazione del database, ma limita la modifica di alcune impostazioni avanzate per la versione gratuita, da noi utilizzata.

L'impossibilità di modificare questo parametro rende molto difficile accedere a dati non aggiornati sui nodi secondari, che si allineano quindi in maniera pressoché istantanea rispetto alle modifiche sul nodo primario.
Questo ha limitato in parte la possibilità di sperimentazione.

Inoltre, in alcune situazioni, avere un nodo secondario in ritardo rispetto agli altri ha dei vantaggi.
Per esempio, se si verificano errori umani o applicativi che corrompono i dati, essendo il nodo secondario ritardato, è possibile usarlo per recuperare una versione precedente e non danneggiata dei dati.
Solitamente, ai nodi secondari caratterizzati da un ritardo di replicazione viene assegnato un livello di priorità pari a 0, il quale evita che vengano eletti come nodo primario.

4.7 Perdita di aggiornamento

La perdita di aggiornamento (lost update) è un'anomalia che si verifica quando due o più transazioni leggono lo stesso valore e poi lo aggiornano basandosi sullo stesso valore ottenuto dalla lettura. Il risultato finale è che uno degli aggiornamenti viene sovrascritto dall'altro, portando alla perdita di uno dei due aggiornamenti.

Un esempio classico di perdita di aggiornamento è quello in cui, in contemporanea, due transazioni leggono lo stesso valore iniziale e poi aggiornano il prezzo con un incremento.
I due script *perdita_aggiornamento_t*1 e *perdita_aggiornamento_t*2, di seguito riportati, emulano questo scenario, se eseguiti in contemporanea in due terminali distinti.
Infatti, il fatto che ciascuna transazione si ponga in attesa, dopo aver letto il prezzo del capo da modificare, lascia spazio all'altra transazione di leggere lo stesso valore. Dopo l'attesa, entrambe le transazioni cercano di aggiornare il prezzo con un incremento; una delle due potrebbe sovrascrivere l'altra, portando a una perdita di aggiornamento.
Tuttavia, anche con il minimo livello di isolamento specificato come *writeConcern*, la seconda transazione che tenta di eseguire l'update va incontro ad un errore:
MongoServerError[WriteConflict]: Caused by :: Write conflict during plan execution and yielding is disabled. :: Please retry your operation or multi-document transaction.
L'errore *MongoServerError[WriteConflict]* indica che c'è stato un conflitto durante l'esecuzione del piano di scrittura: MongoDB rileva che due transazioni stanno cercando di aggiornare lo stesso documento e una transazione deve essere abortita per evitare incoerenze.
Quando si verifica un conflitto di scrittura, MongoDB segnala l'errore di conflitto di scrittura e consiglia di riprovare l'operazione. Le transazioni che non riescono a causa di conflitti devono essere ripetute e questo è generalmente gestito a livello di applicazione, grazie all'uso del meccanismo delle callback.

Questo comportamento di MongoDB garantisce che anomalie di perdita di aggiornamento non possano verificarsi poiché è in grado di rilevare confitti.

In figura 4.1 è riportato uno schema che esemplifica l'ordine con cui le operazioni delle due transazioni sono eseguite.

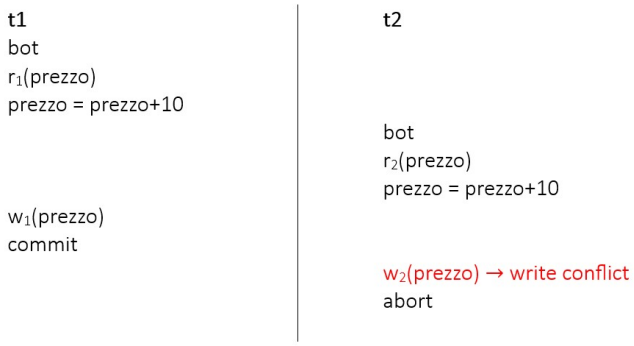

Figura 4.1: Timeline delle operazioni delle due transazioni

```python
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client1 = MongoClient(connection_string)


def callback(session, capo_id=None, additional_price=0):
    myCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    doc = myCollection.find_one(
        {'capoId': capo_id},
        session=session,
    )
    print("Documento da modificare: ", doc)
    initial_price = doc.get("prezzo").to_decimal()
    print("Prezzo iniziale: ", initial_price)

    time.sleep(3)

    try:
        myCollection.update_one({'capoId': capo_id},
                            {"$set": {"prezzo": Decimal128(initial_price + additional_price)}},
                            session=session)

        modified_doc = myCollection.find_one(
            {'capoId': capo_id},
            session=session,
        )
        final_price = modified_doc.get("prezzo").to_decimal()
        print("\n\nDocumento modificato: ", modified_doc)
        print("Prezzo finale: ", final_price)

        try:
            session.commit_transaction()
            print("\n\nTransazione andata a buon fine.\n")
        except Exception as e:
            print(f"\n\nErrore durante il commit della transazione: {e.args[0]}")
            session.abort_transaction()
            return

    except Exception as e:
        print(f"\n\nErrore durante l'aggiornamento: {e.args[0]}")
        print("\n\nTransazione abortita. \n")
        session.abort_transaction()
        return


def callback_wrapper(s):
    callback(
```

```
        session=s,
        capo_id=1,
        additional_price=10
    )


with client1.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

client1.close()
```

Listing 4.14: Perdita di aggiornamento sul terminale 1

```
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client2 = MongoClient(connection_string)


def callback(session, capo_id=None, additional_price=0):
    myCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    doc = myCollection.find_one(
        {'capoId': capo_id},
        session=session,
    )
    print("Documento da modificare: ", doc)
    initial_price = doc.get("prezzo").to_decimal()
    print("Prezzo iniziale: ", initial_price)

    time.sleep(3)

    try:
        myCollection.update_one({'capoId': capo_id},
                        {"$set": {"prezzo": Decimal128(initial_price + additional_price)}},
                        session=session)

        modified_doc = myCollection.find_one(
            {'capoId': capo_id},
            session=session,
        )
        final_price = modified_doc.get("prezzo").to_decimal()
        print("\n\nDocumento modificato: ", modified_doc)
        print("Prezzo finale: ", final_price)

        try:
            session.commit_transaction()
            print("\n\nTransazione andata a buon fine.\n")
        except Exception as e:
            print(f"\n\nErrore durante il commit della transazione: {e.args[0]}")
            session.abort_transaction()

    except Exception as e:
        print(f"\n\nErrore durante l'aggiornamento: {e.args[0]}")
        print("\n\nTransazione abortita. \n")

    return


def callback_wrapper(s):
    callback(
```

```
55          session=s,
56          capo_id=1,
57          additional_price=10
58      )
59
60
61  with client2.start_session() as session:
62      try:
63          session.with_transaction(
64              callback_wrapper,
65              read_concern=ReadConcern(level="local"),
66              write_concern=WriteConcern(w=1, j=False)
67          )
68      except PyMongoError as e:
69          print(f"Transazione fallita: {e.args[0]}")
70
71  client2.close()
```

Listing 4.15: Perdita di aggiornamento sul terminale 2

Per una maggiore comprensione, si è costruita la timeline delle operazioni di scrittura, mostrata in figura 4.2. Con due differenti colori si distinguono le operazioni che determinano le due transazioni concorrenti.

1. Al tempo t_{10}, il nodo primario applica la scrittura richiesta dalla transazione t_1. I dati sul nodo primario riflettono la scrittura appena avvenuta. I dati su entrambi nodi secondari riflettono lo stato determinato da una precedente scrittura.

2. Al tempo t_{20}, il nodo primario riceve la scrittura richiesta dalla transazione t_2 e il server mongos rileva un conflitto di scrittura, impedendo alla transazione di procedere.

3. Al tempo t_{11}, il nodo secondario 1 applica la scrittura richiesta dalla transazione t_1.

4. Al tempo t_{12}, il nodo secondario 2 applica la scrittura richiesta dalla transazione t_2.

5. Al tempo t_{13}, il nodo primario riceve l'ack da parte del nodo secondario 1 e invia l'acknowlwdge al client.

6. Al tempo t_{14}, il nodo primario riceve l'ack da parte del nodo secondario 2.

7. Al tempo t_{15}, il nodo secondario 1 riceve l'avviso, attraverso il meccanismo di replica, di aggiornare il suo snapshot con la scrittura più recentemente confermata (w: "majority").

8. Al tempo t_{16}, il nodo secondario 2 riceve l'avviso, attraverso il meccanismo di replica, di aggiornare il suo snapshot con la scrittura più recentemente confermata (w: "majority").

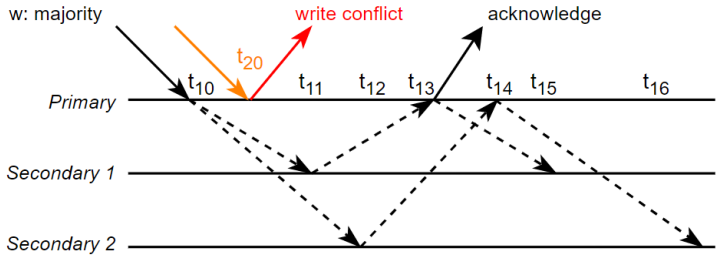

Figura 4.2: Timeline delle operazioni di scrittura

4.7.1 Inserimento di uno scontrino

Per tentare di emulare una perdita di aggiornamento si è fatto un ulteriore tentativo, più articolato, che consiste nell'inserimento di uno scontrino da parte di due transazioni. L'acquisto di prodotti, quindi l'inserimento di uno scontrino, presuppone che si vada non solo a creare un nuovo documento nella collezione scontrini, ma che si vada a modificare la quantità dei prodotti acquistati, decrementandone quindi il numero. Si tratta quindi di transazioni multi-documento.
Le due transazioni di seguito riportate consistono nell'acquisto di un cappotto nel primo caso e di un cappotto e un paio di jeans nel secondo caso. Entrambe le transazioni presuppongo quindi di modificare la quantità di uno stesso capo. Una perdita di aggiornamento si potrebbe verificare qualora entrambe le transazioni effettuassero un decremento dell'unità venduta, a partire dallo stesso valore iniziale, archiviando così una quantità che non sarebbe fedele al verificarsi delle due vendite. Nel momento in cui però la seconda transazione, fatta partire successivamente alla prima, tenta di aggiornare la quantità del cappotto, questa va in abort, poiché il valore di quantità che ha letto e che tenta di modificare è stato già modificato dalla transazione 1.
Come nell'esempio precedente, si verifica una situazione di conflitto di scrittura che impedisce ad una delle due transazioni di procedere, impedendo il verificarsi dell'anomalia.

I due script di seguito riportati, *inserimento_scontrino_t1* e *inserimento_scontrino_t2*, riproducono la situazione appena descritta, se eseguiti in contemporanea su due terminali distinti, immediatamente uno di seguito all'altro.

```
1  from datetime import datetime
2
3  from bson import Decimal128
4  from pymongo import MongoClient, WriteConcern
5  from pymongo.errors import PyMongoError
6  from pymongo.read_concern import ReadConcern
7  import time
8
9  connection_string = "..."
10 client1 = MongoClient(connection_string)
11
12
13 def callback(session, capi, taglie):
14     capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
```

```
        write_concern=WriteConcern(w=1, j=False)
    )
    scontriniCollection = session.client.negozio_abbigliamento.scontrini.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    articoli = []
    totale_complessivo = 0

    print("")
    for capo, taglia in zip(capi, taglie):
        articolo = capiCollection.find_one({'nome': capo}, {'_id': False}, session=session)
        prezzo_articolo = articolo.get("prezzo").to_decimal()

        print(f"Tentativo di acquisto di {articolo.get('nome')} (taglia: {taglia}) "
              f"con ID {articolo.get('capoId')}")
        print(articolo)

        time.sleep(3)

        capiCollection.update_one({'nome': capo}, {'$inc': {f'disponibilita.{taglia}': -1}},
                                  session=session)
        articolo_aggiornato = capiCollection.find_one({'nome': capo}, {'_id': False},
                                                      session=session)
        print(f"Aggiornamento quantità: {articolo_aggiornato}\n")

        articoli.append({"nome": capo, "quantita": 1, "prezzo_totale":
                         Decimal128(str(prezzo_articolo)),"taglia": taglia})
        totale_complessivo += prezzo_articolo

    totale_complessivo = Decimal128(str(totale_complessivo))

    nuovo_scontrino = {
        "data": datetime.strptime(str(datetime.now().date()), "%Y-%m-%d"),
        "articoli": articoli,
        "totale_complessivo": totale_complessivo
    }

    try:
        scontriniCollection.insert_one(nuovo_scontrino)
        print("Acquisto terminato.")
        print(f"Scontrino: {nuovo_scontrino}\n")
        time.sleep(4)
        session.commit_transaction()
        print("Transazione andata a buon fine.\n")
        time.sleep(3)

    except Exception as e:
        print(f"\n\nErrore durante la transazione: {e.args[0]}")
        session.abort_transaction()
        print("\n\nTransazione abortita. \n")
    return


def callback_wrapper(s):
    capi = ["Cappotto"]
    taglie = ["M"]
    callback(
        session=s,
        capi=capi,
        taglie=taglie
    )


with client1.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

```

```
client1.close()
```

Listing 4.16: Inserimento scontrino sul terminale 1

```
from datetime import datetime

from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client2 = MongoClient(connection_string)


def callback(session, capi, taglie):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )
    scontriniCollection = session.client.negozio_abbigliamento.scontrini.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    articoli = []
    totale_complessivo = 0

    print("")
    for capo, taglia in zip(capi, taglie):
        articolo = capiCollection.find_one({'nome': capo}, {'_id': False}, session=session)
        prezzo_articolo = articolo.get("prezzo").to_decimal()

        print(f"Tentativo di acquisto di {articolo.get('nome')} (taglia: {taglia}) "
              f"con ID {articolo.get('capoId')}")
        print(articolo)

        time.sleep(1)

        capiCollection.update_one({'nome': capo}, {'$inc': {f'disponibilita.{taglia}': -1}},
                                  session=session)
        articolo_aggiornato = capiCollection.find_one({'nome': capo}, {'_id': False},
                                                      session=session)
        print(f"Aggiornamento quantità: {articolo_aggiornato}\n")

        articoli.append({"nome": capo, "quantita": 1, "prezzo_totale":
                         Decimal128(str(prezzo_articolo)), "taglia": taglia})
        totale_complessivo += prezzo_articolo

    totale_complessivo = Decimal128(str(totale_complessivo))

    nuovo_scontrino = {
        "data": datetime.strptime(str(datetime.now().date()), "%Y-%m-%d"),
        "articoli": articoli,
        "totale_complessivo": totale_complessivo
    }

    try:
        scontriniCollection.insert_one(nuovo_scontrino)
        print("Acquisto terminato.")
        print(f"Scontrino: {nuovo_scontrino}\n")
        time.sleep(4)
        session.commit_transaction()
        print("Transazione andata a buon fine.\n")
        time.sleep(3)

    except Exception as e:
        print(f"\n\nErrore durante la transazione: {e.args[0]}")
        session.abort_transaction()
        print("\n\nTransazione abortita. \n")
    return


def callback_wrapper(s):
    capi = ["Cappotto", "Jeans"]
```

```
71      taglie = ["M", "L"]
72      callback(
73          session=s,
74          capi=capi,
75          taglie=taglie
76      )
77
78
79  with client2.start_session() as session:
80      try:
81          session.with_transaction(
82              callback_wrapper,
83              read_concern=ReadConcern(level="local"),
84              write_concern=WriteConcern(w=1, j=False)
85          )
86      except PyMongoError as e:
87          print(f"Transazione fallita: {e.args[0]}")
88
89  client2.close()
```

Listing 4.17: Inserimento scontrino sul terminale 2

4.8 Letture inconsistenti

L'anomalia letture inconsistenti si verifica quando, all'interno di una transazione, una lettura dello stesso dato produce risultati diversi in due momenti diversi. Questo può portare a interpretazioni errate dei dati a causa della mancanza di coerenza: nell'ambito di una transazione non deve essere possibile leggere valori differenti.

Una situazione d'esempio per la perdita di aggiornamento è quella in cui c'è una prima transazione che legge una certa variabile e poco dopo la seconda legge e aggiorna la stessa variabile andando in commit. A questo punto la prima transazione rilegge la stessa variabile, ma a questo punto vede il valore aggiornato. Quindi, nell'ambito della stessa transazione, vengono letti due valori per la stessa variabile, cosa che non dovrebbe accadere mai.

I due script di seguito riportati, *letture_inconsistenti_t1* e *letture_inconsistenti_t2*, cercano di riprodurre la situazione appena descritta. I due script devono essere eseguiti in contemporanea su due terminali distinti. Il codice è stato scritto in modo che si crei il giusto alternarsi delle istruzioni attraverso dei *time sleep*.
La prima transazione legge il valore per il prezzo del capo di abbigliamento con *capoId* pari a 1, corrispondente a 19.99. La seconda transazione, legge lo stesso valore, lo aggiorna aumentandolo di 5. Nel frattempo la prima transazione legge di nuovo il valore, trovando un prezzo coerente, ovvero 19.99. A questo punto la seconda transazione va in commit e, infine, la prima transazione legge per la terza volta il valore, trovando ancora una volta il valore corretto.
Non si riesce a leggere un valore inconsistente.

In figura 4.3 è riportato uno schema riassuntivo della sequenza di esecuzione delle due transazioni.

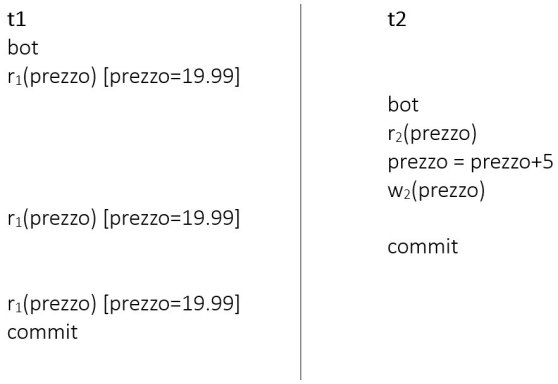

Figura 4.3: Timeline delle operazioni delle due transazioni

```python
import time

from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern

connection_string = "..."

client1 = MongoClient(connection_string)


def callback(session, capo_id=None):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        read_concern=ReadConcern("local"),
        write_concern=WriteConcern(w=1, j=False)
    )

    for i in range(0,3):
        doc = capiCollection.find_one(
            {'capoId': capo_id},
            {'_id': False},
            session=session,
        )
        print(doc)
        initial_price = doc.get("prezzo").to_decimal()
        print("Prezzo del capo: ", initial_price)
        print("")
        time.sleep(3)

    try:
        session.commit_transaction()
        print("\n\nTransazione andata a buon fine.\n")
    except Exception as e:
        print(f"\n\nErrore durante il commit della transazione: {e.args[0]}")
        session.abort_transaction()
        return


def callback_wrapper(s):
    callback(
        session=s,
        capo_id=1,
    )


with client1.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")
```

```
55
56 client1.close()
```

Listing 4.18: Tentativo di lettura inconsistente sul terminale 1

```
1  from bson import Decimal128
2  from pymongo import MongoClient, WriteConcern
3  from pymongo.errors import PyMongoError
4  from pymongo.read_concern import ReadConcern
5  from pymongo.read_preferences import ReadPreference
6
7  import time
8
9  connection_string = "..."
10
11 client1 = MongoClient(connection_string)
12
13
14 def callback(session, capo_id=None, additional_price=0):
15     capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
16         read_concern=ReadConcern("local"),
17         write_concern=WriteConcern(w=1, j=False)
18     )
19
20     doc = capiCollection.find_one(
21         {'capoId': capo_id},
22         {'_id': False},
23         session=session,
24     )
25     print("Documento da modificare: ", doc)
26     initial_price = doc.get("prezzo").to_decimal()
27     print("Prezzo iniziale: ", initial_price)
28
29     try:
30         capiCollection.update_one({'capoId': capo_id},
31                                   {"$set": {"prezzo": Decimal128(initial_price + additional_price)}},
32                                   session=session)
33
34         modified_doc = capiCollection.find_one(
35             {'capoId': capo_id},
36             {'_id': False},
37             session=session,
38         )
39         final_price = modified_doc.get("prezzo").to_decimal()
40         print("\nDocumento modificato: ", modified_doc)
41         print("Prezzo finale: ", final_price)
42
43         time.sleep(2)
44
45         try:
46             session.commit_transaction()
47             print("\n\nTransazione andata a buon fine.\n")
48         except Exception as e:
49             print(f"\n\nErrore durante il commit della transazione: {e.args[0]}")
50             session.abort_transaction()
51             return
52
53     except Exception as e:
54         print(f"\n\nErrore durante l'aggiornamento: {e.args[0]}")
55         print("\nTransazione abortita. \n")
56         session.abort_transaction()
57         return
58
59
60 def callback_wrapper(s):
61     callback(
62         session=s,
63         capo_id=1,
64         additional_price=5
65     )
66
67
68 with client1.start_session() as session:
69     try:
```

```
70          session.with_transaction(
71              callback_wrapper,
72              read_concern=ReadConcern(level="local"),
73              write_concern=WriteConcern(w=1, j=False)
74          )
75      except PyMongoError as e:
76          print(f"Transazione fallita: {e.args[0]}")
77
78  client1.close()
```

Listing 4.19: Tentativo di lettura inconsistente sul terminale 2

4.9 Lettura sporca

La lettura sporca (dirty read) è un'anomalia che si verifica quando una transazione legge dati che sono stati modificati da un'altra transazione che però non è ancora stata confermata (commit), quindi l'effetto della transazione non è garantito che divenga definitivo e durevole. Infatti, se la transazione che ha effettuato le modifiche viene annullata (abort), l'altra transazione si basa su dati che non esistono e questo potrebbe determinare delle inconsistenze.
In altre parole, su un terminale si avvia una transazione, che modifica un documento e non conferma immediatamente la transazione, con l'obiettivo di vedere se un'altra transazione può leggere questo stato non confermato. Su un secondo terminale, si avvia quindi una seconda transazione che legge il documento modificato dalla prima sessione per osservare se rileva le modifiche non confermate (lettura sporca) o no.

Di default, MongoDB non permette letture sporche, utilizzando meccanismi di coerenza e replica per garantire che le letture siano consistenti e affidabili. Le transazioni distribuite e l'utilizzo di replica set impongono che le letture avvengano su dati confermati.
Tuttavia, nonostante la configurazione predefinita di MongoDB impedisca le letture sporche, ci sono alcuni scenari in cui qualcosa di simile a questo comportamento potrebbe accadere.
Con un Read Concern di livello "local", una transazione legge i dati locali al nodo corrente senza garanzia che questi siano replicati o confermati dagli altri nodi. Se si verificasse un guasto, prima che la transazione sia completamente replicata e confermata nel replica set, allora potrebbe essere necessario eseguire un rollback della transazione, con conseguente lettura di dati "sporchi" da parte della transazione. Con il meccanismo di rollback comunque anche la transazione che ha letto dati non confermati sarebbe annullata, ripristinando uno stato consistente.

In estrema sintesi, poiché in MongoDB le modifiche non confermate all'interno di una transazione non sono visibili ad altre transazioni fino a quando la transazione non è stata confermata, non è possibile che le letture da parte di altre transazioni concorrenti vedano dati non confermati.

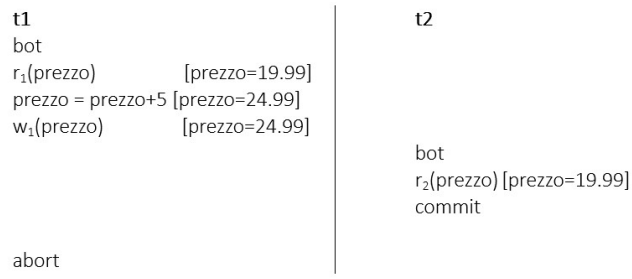

Figura 4.4: Timeline delle operazioni delle due transazioni

I due script di seguito riportati, *lettura_sporca_t1* e *lettura_sporca_t2*, riproducono la situazione appena descritta. I due script devono essere eseguiti in contemporanea su due terminali distinti, immediatamente uno di seguito all'altro.

```python
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
from pymongo.read_preferences import ReadPreference

import time

connection_string = "..."

client1 = MongoClient(connection_string)


def callback(session, capo_id=None, additional_price=0):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        read_concern=ReadConcern("local"),
        write_concern=WriteConcern(w=1, j=False)
    )

    doc = capiCollection.find_one(
        {'capoId': capo_id},
        {'_id': False},
        session=session,
    )
    print("Documento da modificare: ", doc)
    initial_price = doc.get("prezzo").to_decimal()
    print("Prezzo iniziale: ", initial_price)

    try:
        capiCollection.update_one({'capoId': capo_id},
                        {"$set": {"prezzo": Decimal128(initial_price + additional_price)}},
                        session=session)

        modified_doc = capiCollection.find_one(
            {'capoId': capo_id},
            {'_id': False},
            session=session,
        )
        final_price = modified_doc.get("prezzo").to_decimal()
        print("\nDocumento modificato: ", modified_doc)
        print("Prezzo finale: ", final_price)

        time.sleep(15)

        session.abort_transaction()
        print("\n\nTransazione abortita. \n")

    except Exception as e:
        print(f"\n\nErrore durante l'aggiornamento: {e.args[0]}")
        session.abort_transaction()
        print("\n\nTransazione abortita. \n")
```

```
52
53      return
54
55
56  def callback_wrapper(s):
57      callback(
58          session=s,
59          capo_id=1,
60          additional_price=5
61      )
62
63
64  with client1.start_session() as session:
65      try:
66          session.with_transaction(
67              callback_wrapper,
68              read_concern=ReadConcern(level="local"),
69              write_concern=WriteConcern(w=1, j=False)
70          )
71      except PyMongoError as e:
72          print(f"Transazione fallita: {e.args[0]}")
73
74  client1.close()
```

Listing 4.20: Tentativo di lettura sporca sul terminale 1

```
1   from pymongo import MongoClient, WriteConcern
2   from pymongo.errors import PyMongoError
3   from pymongo.read_concern import ReadConcern
4
5   connection_string = "..."
6   client2 = MongoClient(connection_string)
7
8
9   def callback(session, capo_id=None):
10      capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
11          read_concern=ReadConcern("local"),
12          write_concern=WriteConcern(w=1, j=False)
13      )
14
15      doc = capiCollection.find_one(
16          {'capoId': capo_id},
17          {'_id': False},
18          session=session,
19      )
20      print(doc)
21      initial_price = doc.get("prezzo").to_decimal()
22      print("Prezzo del capo: ", initial_price)
23
24      try:
25          session.commit_transaction()
26          print("\n\nTransazione andata a buon fine.\n")
27      except Exception as e:
28          print(f"\n\nErrore durante il commit della transazione: {e.args[0]}")
29          session.abort_transaction()
30          return
31
32
33
34  def callback_wrapper(s):
35      callback(
36          session=s,
37          capo_id=1,
38      )
39
40
41  with client2.start_session() as session:
42      try:
43          session.with_transaction(
44              callback_wrapper,
45              read_concern=ReadConcern(level="local"),
46              write_concern=WriteConcern(w=1, j=False)
47          )
48      except PyMongoError as e:
```

```
49         print(f"Transazione fallita: {e.args[0]}")
50
51 client2.close()
```

Listing 4.21: Tentativo di lettura sporca sul terminale 2

4.10 Inserimento fantasma

L'inserimento fantasma, anche chiamato "phantom reads", è una situazione anomala in cui una transazione, durante la sua esecuzione, osserva un insieme di righe che cambia perché un'altra transazione ha inserito, aggiornato o cancellato righe che soddisfano i criteri della query originale della prima transazione.

Un classico esempio di questo fenomeno è quando la prima transazione conta il numero di documenti di una collezione. La seconda transazione aggiunge un ulteriore documento, che quindi altera il conteggio. Quando la prima transazione riesegue l'operazione di conteggio, ottiene un valore diverso dal precedente, che ovviamente, come nel caso della lettura inconsistente, è da evitare.

Gli script di seguito riportati, *inserimento_fantasma_t1* e *inserimento_fantasma_t2*, cercano di riprodurre tale anomalia.
In particolare, la prima transazione conta e stampa a video il numero di capi di colore rosso presenti nella collezione *capi_abbigliamento*, pari a 1. A questo punto la seconda transazione vi inserisce un capo di colore rosso. La prima transazione riesegue il conteggio, trovando lo stesso numero di capi letti in precedenza, ovvero 1. La seconda transazione va in commit. La prima, anche dopo il commit, riesegue il conteggio e ottiene nuovamente 1. Sul secondo terminale, inoltre, dopo che la transazione è andata in commit, si esegue il conteggio per verificare che l'inserimento sia correttamente avvenuto.

Lo schema in figura 4.5 mostra l'alternarsi delle operazioni delle due transazioni.

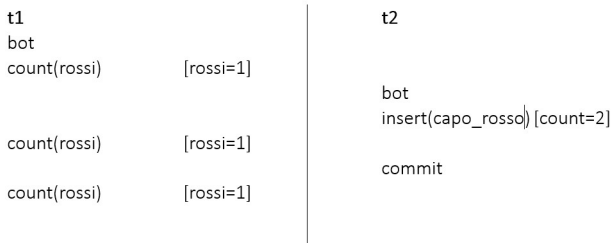

Figura 4.5: Timeline delle operazioni delle due transazioni

```
1 import pprint
2 import time
3
```

```python
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern

connection_string = "..."
client = MongoClient(connection_string)

def callback(session, colore):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    for i in range(0,2):
        print(f"Articoli di colore {colore}.")
        cursor = capiCollection.find({'colore': colore}, {'_id': False}, session=session)
        num_docs = 0
        for document in cursor:
            num_docs += 1
            pprint.pprint(document)
            print()
        print(f"Numero dei capi di colore {colore}: {str(num_docs)}\n")
        time.sleep(5)

    try:
        session.commit_transaction()

    except Exception as e:
        print(f"Errore durante il commit della transazione: {e.args[0]}")
        session.abort_transaction()

def callback_wrapper(s):
    callback(
        session=s,
        colore="rosso"
    )


with client.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

client.close()
```

Listing 4.22: Tentativo di inserimento fantasma sul terminale 1

```python
import pprint

from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern

connection_string = "..."
client = MongoClient(connection_string)

def callback(session, colore):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )


    cursor = capiCollection.find({'colore': colore}, {'_id': False}, session=session)
    print(f"Articoli di colore {colore}.")
    num_docs = 0
    for document in cursor:
        num_docs += 1
        pprint.pprint(document)
        print()
```

```
24      print(f"Numero dei capi di colore {colore}: {str(num_docs)}.\n\n")
25      try:
26          capiCollection.insert_one({"nome": "Maglione", "prezzo": Decimal128("59.99"),
27                                     "colore": "rosso", "disponibilita": {"S": 30, "M": 50, "L": 20}})
28          print(f"Inserimento di un nuovo articolo di colore {colore}.\n")
29
30      except Exception as e:
31          print(f"Errore durante l'inserimento dell'articolo: {e.args[0]}")
32
33      try:
34          session.commit_transaction()
35          print("Transazione andata a buon fine.\n\n")
36
37          print(f"Articoli di colore {colore}.")
38          cursor = capiCollection.find({'colore': colore}, session=session)
39          num_docs = 0
40          for document in cursor:
41              num_docs += 1
42              pprint.pprint(document)
43              print()
44          print(f"Numero dei capi di colore {colore}: {str(num_docs)}.")
45
46      except Exception as e:
47          print(f"Errore durante il commit della transazione: {e.args[0]}")
48          session.abort_transaction()
49
50  def callback_wrapper(s):
51      colore = "rosso"
52      callback(
53          session=s,
54          colore=colore
55      )
56
57
58  with client.start_session() as session:
59      try:
60          session.with_transaction(
61              callback_wrapper,
62              read_concern=ReadConcern(level="local"),
63              write_concern=WriteConcern(w=1, j=False)
64          )
65      except PyMongoError as e:
66          print(f"Transazione fallita: {e.args[0]}")
67
68  client.close()
```

Listing 4.23: Tentativo di inserimento fantasma sul terminale 2

4.11 Aggiornamento fantasma

L'aggiornamento fantasma è un'anomalia che si verifica quando una transazione accede a un insieme di dati che sono soggetti ad una condizione specifica e, nel mentre, un'altra transazione modifica alcuni dati in modo tale che l'insieme dei dati che soddisfano la condizione siano differenti. Se la prima transazione accede all'insieme dei dati con letture che sono inframezzate dalla seconda transazione, allora la prima transazione potrebbe leggere un'insieme di dati che non soddisfano la condizione perché l'altra transazione ha modificato alcuni di questi dati.

Si supponga che tra i capi di abbigliamento sia presente un abito composto da giacca e pantalone e si supponga anche che sia possibile vendere l'abito spezzato, quindi solo la giacca o solo il pantalone, oltre che come completo. Tuttavia, si suppone che esista il vincolo per cui il prezzo dell'abito completo sia inferiore alla

somma dei prezzi di giacca e pantalone venduti separatamente.
I due script di seguito riportati *aggiornamento_fantasma_t1.py* e *aggiornamento_fantasma_t1.py* rappresentano un tentativo di emulare un aggiornamento fantasma in uno scenario di questo tipo.
La transazione che viene eseguita con il primo script legge il prezzo della giacca e poi si pone in attesa. Nel frattempo la seconda transazione legge i prezzi degli articoli e aggiorna il prezzo dei pantaloni e della giacca, in particolare decrementa il prezzo dei pantaloni di 40 € e aumenta il prezzo della giacca della stessa quantità, rispettando complessivamente il vincolo ma cambiando i prezzi individuali. Se a questo punto la prima transazione leggesse il nuovo prezzo dei pantaloni il prezzo risultante dalla somma dei prezzi dei due capi violerebbe il vincolo. Tuttavia, pur usando il minor livello di isolamento, è garantito che la prima transazione acceda ad uno stato consistente, quindi legga il prezzo della giacca precedente alla modifica. Ciò significa che, anche se un'altra transazione modifica i dati nel mezzo, la prima transazione vedrà uno stato coerente fino alla fine.

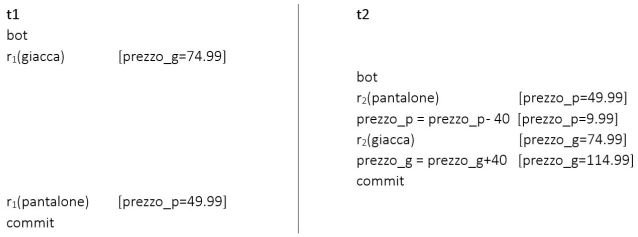

Figura 4.6: Timeline delle operazioni delle due transazioni

```python
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client = MongoClient(connection_string)

def callback(session):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        read_concern=ReadConcern(level="local"),
        write_concern=WriteConcern(w=1, j=False)
    )

    try:

        giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id':False}, session=session)
        prezzo_giacca = giacca.get("prezzo").to_decimal()

        print("Prezzo giacca (prima della modifica di T2): ", prezzo_giacca)
        print("\n\n")

        time.sleep(5)

        abito = capiCollection.find_one({'nome': 'Abito'}, {'_id':False}, session=session)
        prezzo_abito = abito.get("prezzo").to_decimal()

        pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, {'_id':False}, session=
    session)
```

```
30          prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
31
32          prezzo_completo = prezzo_giacca + prezzo_pantaloni
33
34          print("Prezzo pantaloni (dopo le modifiche di T2): ", prezzo_pantaloni)
35          print("\nPrezzo completo giacca e pantaloni (dopo le modifiche di T2): ", Decimal128(
        prezzo_completo))
36          print("Prezzo abito: ", prezzo_abito)
37          print("")
38
39          if prezzo_completo >= prezzo_abito:
40              print("Vincolo soddisfatto")
41          else:
42              session.abort_transaction()
43              print("Vincolo non soddisfatto")
44              print("Transazione abortita.")
45
46      except Exception as e:
47          print(f"Errore durante la transazione: {e.args[0]}")
48          session.abort_transaction()
49
50      finally:
51          session.end_session()
52
53
54  def callback_wrapper(s):
55      callback(
56          session=s
57      )
58
59
60  with client.start_session() as session:
61      try:
62          session.with_transaction(
63              callback_wrapper,
64              read_concern=ReadConcern(level="local"),
65              write_concern=WriteConcern(w=1, j=False)
66          )
67      except PyMongoError as e:
68          print(f"Transazione fallita: {e.args[0]}")
69
70  client.close()
```

Listing 4.24: Tentativo di aggiornamento fantasma sul terminale 1

```
1   from bson import Decimal128
2   from pymongo import MongoClient, WriteConcern
3   from pymongo.errors import PyMongoError
4   from pymongo.read_concern import ReadConcern
5   import time
6
7   connection_string = "..."
8   client = MongoClient(connection_string)
9
10  def callback(session):
11      capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
12          read_concern=ReadConcern(level="local"),
13          write_concern=WriteConcern(w=1, j=False)
14      )
15
16      try:
17
18          abito = capiCollection.find_one({'nome': 'Abito'}, {'_id': False}, session=session)
19          prezzo_abito = abito.get("prezzo").to_decimal()
20
21          pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, {'_id': False}, session=
        session)
22          prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
23
24          giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id':False}, session=session)
25          prezzo_giacca = giacca.get("prezzo").to_decimal()
26
27          prezzo_completo = prezzo_giacca + prezzo_pantaloni
28
```

```
        print("Prezzo giacca prima dell'update: ", prezzo_giacca)
        print("Prezzo pantaloni prima dell'update: ", prezzo_pantaloni)
        print("Prezzo completo giacca e pantaloni: ", round(float(prezzo_completo), 2))
        print("Prezzo abito: ", prezzo_abito)
        print("")

        if prezzo_completo >= prezzo_abito:

            capiCollection.update_one({'nome': 'Pantaloni'}, {'$set': {'prezzo':
                                       Decimal128(prezzo_pantaloni-40)}}, session=session)

            pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, session=session)
            prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()

            print("Prezzo pantaloni dopo l'update: ", prezzo_pantaloni)
            print("")

            capiCollection.update_one({'nome': 'Giacca'},
                                      {'$set': {'prezzo': Decimal128(prezzo_giacca+40)}},
                                      session=session)

            giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id': False}, session=session
    )
            prezzo_giacca = giacca.get("prezzo").to_decimal()
            print("Prezzo giacca dopo l'update: ", prezzo_giacca)
            print("")

            session.commit_transaction()
            print("Commit")
            print("")
        else:
            session.abort_transaction()
            print("Abort: il prezzo del completo deve superare quello dell'abito")
            print("")

    except Exception as e:
        print(f"Errore durante la transazione: {e.args[0]}")
        session.abort_transaction()

    finally:
        session.end_session()

def callback_wrapper(s):
    callback(
        session=s
    )


with client.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

client.close()
```

Listing 4.25: Tentativo di aggiornamento fantasma sul terminale 2

4.12 Write Skew

L'anomalia write skew è una situazione che si verifica quando due o più transazioni effettuano operazioni di lettura e scrittura sui dati, che si ordinano in modo tale da rendere il risultato inconsistente con i vincoli imposti.

Una classica situazione è ambientata nell'ambito medico. Supponiamo che nella collezione *dottori* ci sia un vincolo per cui almeno uno dei documenti abbia il campo *a lavoro* valorizzato con *true*. Prima dell'inizio della transazione ci sono solo due documenti nella collezione, entrambi con il campo *a lavoro* valorizzato a true. La prima transazione inizia e modifica in *false* il campo *a lavoro* del dottore numero 1. La seconda fa lo stesso per il dottore numero 2. Entrambe le transazioni, localmente, hanno modificato la collezione in modo da mantenere il vincolo, ma, nel complesso, il vincolo non è più rispettato, perché le due transazioni non hanno visto la modifica apportata dall'altra.
Nell'esempio di seguito riportato viene mostrato il verificarsi di questa anomalia. Si immagini che nella nostra applicazione esista un vincolo per cui la somma del prezzo della giacca e del pantalone deve esser sempre maggiore del prezzo del completo.

La prima transazione diminuisce il prezzo della giacca di 20, che non viola il vincolo imposto.
La seconda opera la stessa sottrazione sul pantalone, senza violare il vincolo.
Tuttavia, T1 e T2 vedono lo stesso stato iniziale del database. Entrambe hanno preso una decisione basata su quanto visto, ma le loro scritture determinano uno stato finale che non rispetta il vincolo. Nessuna delle transazioni ha visto lo stato intermedio che includeva la modifica dell'altra.

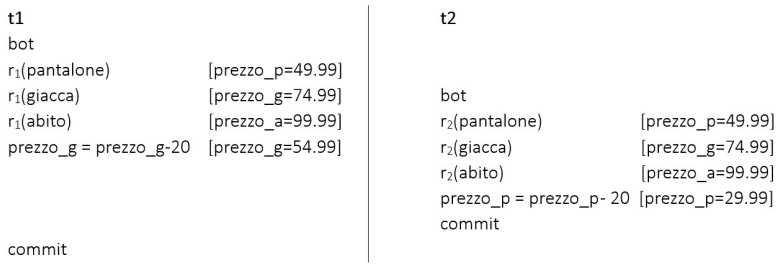

Figura 4.7: Timeline delle operazioni delle due transazioni

```python
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client = MongoClient(connection_string)

def callback(session):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    try:
        abito = capiCollection.find_one({'nome': 'Abito'}, session=session)
        prezzo_abito = abito.get("prezzo").to_decimal()

        giacca = capiCollection.find_one({'nome': 'Giacca'}, session=session)
        prezzo_giacca = giacca.get("prezzo").to_decimal()
```

```
        pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, session=session)
        prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()

        prezzo_completo = prezzo_giacca + prezzo_pantaloni

        print("Prezzo giacca prima dell'update: ", prezzo_giacca)
        print("Prezzo pantaloni prima dell'update: ", prezzo_pantaloni)
        print("Prezzo completo giacca e pantaloni prima dell'update: ", prezzo_completo)
        print("Prezzo cappotto prima dell'update: ", prezzo_abito)
        print("")

        if prezzo_completo >= prezzo_abito + 20:
            capiCollection.update_one({'nome': 'Giacca'}, {'$set': {'prezzo':
                                    Decimal128(str(prezzo_giacca-20))}}, session=session)

            giacca = capiCollection.find_one({'nome': 'Giacca'}, session=session)
            prezzo_giacca = giacca.get("prezzo").to_decimal()
            pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, session=session)
            prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
            prezzo_completo = prezzo_giacca + prezzo_pantaloni

            abito = capiCollection.find_one({'nome': 'Abito'}, session=session)
            prezzo_abito = abito.get("prezzo").to_decimal()

            print("Prezzo giacca dopo l'update: ", prezzo_giacca)
            print("Prezzo pantaloni dopo l'update: ", prezzo_pantaloni)
            print("Prezzo completo giacca e pantaloni dopo l'update: ", prezzo_completo)
            print("Prezzo cappotto dopo l'update: ", prezzo_abito)
            time.sleep(3)
            session.commit_transaction()
            print("\nTransazione andata a buon fine.\n")
            time.sleep(3)
        else:
            session.abort_transaction()
            print("Abort: il prezzo del completo deve superare quello dell'abito.\n")
            return
    except Exception as e:
        print(f"Errore durante la transazione: {e.args[0]}")
        session.abort_transaction()
        return
    finally:
        session.end_session()

    giacca = capiCollection.find_one({'nome': 'Giacca'})
    prezzo_giacca = giacca.get("prezzo").to_decimal()
    pantaloni = capiCollection.find_one({'nome': 'Pantaloni'})
    prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
    prezzo_completo = prezzo_giacca + prezzo_pantaloni
    abito = capiCollection.find_one({'nome': 'Abito'})
    prezzo_abito = abito.get("prezzo").to_decimal()

    print("Prezzo giacca dopo il commit: ", prezzo_giacca)
    print("Prezzo pantaloni dopo il commit: ", prezzo_pantaloni)
    print("Prezzo cappotto dopo il commit: ", prezzo_abito)
    print("Prezzo completo giacca e pantaloni dopo il commit: ", prezzo_completo)

    if prezzo_completo >= prezzo_abito:
        print("Il vincolo è rispettato")
    else:
        print("Attenzione! Il vincolo non è rispettato")


def callback_wrapper(s):
    callback(
        session=s
    )


with client.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
```

```
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

client.close()
```

Listing 4.26: Tentativo di write skew sul terminale 1

```
from bson import Decimal128
from pymongo import MongoClient, WriteConcern
from pymongo.errors import PyMongoError
from pymongo.read_concern import ReadConcern
import time

connection_string = "..."
client = MongoClient(connection_string)

def callback(session):
    capiCollection = session.client.negozio_abbigliamento.capi_abbigliamento.with_options(
        write_concern=WriteConcern(w=1, j=False)
    )

    try:
        abito = capiCollection.find_one({'nome': 'Abito'}, {'_id':False}, session=session)
        prezzo_abito = abito.get("prezzo").to_decimal()

        giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id':False}, session=session)
        prezzo_giacca = giacca.get("prezzo").to_decimal()
        pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, {'_id':False},
                                            session=session)
        prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()

        prezzo_completo = prezzo_giacca + prezzo_pantaloni

        print("Prezzo giacca prima dell'update: ", prezzo_giacca)
        print("Prezzo pantaloni prima dell'update: ", prezzo_pantaloni)
        print("Prezzo completo giacca e pantaloni prima dell'update: ", prezzo_completo)
        print("Prezzo abito: ", prezzo_abito)
        print("")

        if prezzo_completo >= prezzo_abito + 20:
            capiCollection.update_one({'nome': 'Pantaloni'}, {'$set': {'prezzo':
                                    Decimal128(str(prezzo_pantaloni-20))}}, session=session)

            pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, {'_id':False},
                                                session=session)
            prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
            giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id':False},
                                             session=session)
            prezzo_giacca = giacca.get("prezzo").to_decimal()
            prezzo_completo = prezzo_giacca + prezzo_pantaloni

            abito = capiCollection.find_one({'nome': 'Abito'}, {'_id':False},
                                            session=session)
            prezzo_abito = abito.get("prezzo").to_decimal()

            print("Prezzo giacca dopo l'update: ", prezzo_giacca)
            print("Prezzo pantaloni dopo l'update: ", prezzo_pantaloni)
            print("Prezzo completo giacca e pantaloni dopo l'update: ", prezzo_completo)
            print("Prezzo cappotto dopo l'update: ", prezzo_abito)
            session.commit_transaction()
            print("\nTransazione andata a buon fine.\n")
            time.sleep(3)
        else:
            session.abort_transaction()
            print("Abort: il prezzo del completo deve superare quello dell'abito.\n")
            return
    except Exception as e:
        print(f"Errore durante la transazione: {e.args[0]}")
        session.abort_transaction()
        return
    finally:
        session.end_session()

    giacca = capiCollection.find_one({'nome': 'Giacca'}, {'_id':False})
```

```
    prezzo_giacca = giacca.get("prezzo").to_decimal()
    pantaloni = capiCollection.find_one({'nome': 'Pantaloni'}, {'_id':False})
    prezzo_pantaloni = pantaloni.get("prezzo").to_decimal()
    prezzo_completo = prezzo_giacca + prezzo_pantaloni
    abito = capiCollection.find_one({'nome': 'Abito'}, {'_id':False})
    prezzo_abito = abito.get("prezzo").to_decimal()

    print("Prezzo giacca dopo il commit: ", prezzo_giacca)
    print("Prezzo pantaloni dopo il commit: ", prezzo_pantaloni)
    print("Prezzo cappotto dopo il commit: ", prezzo_abito)
    print("Prezzo completo giacca e pantaloni dopo il commit: ", prezzo_completo)

    if prezzo_completo >= prezzo_abito:
        print("Il vincolo è rispettato")
    else:
        print("Attenzione! Il vincolo non è rispettato")

def callback_wrapper(s):
    callback(
        session=s,
    )


with client.start_session() as session:
    try:
        session.with_transaction(
            callback_wrapper,
            read_concern=ReadConcern(level="local"),
            write_concern=WriteConcern(w=1, j=False)
        )
    except PyMongoError as e:
        print(f"Transazione fallita: {e.args[0]}")

client.close()
```

Listing 4.27: Tentativo di write skew sul terminale 2

Per prevenire il verificarsi di questa anomalia, si sono modificati gli script appena discussi.
Si è introdotto un aggiornamento fittizio che riguarda l'insieme dei dati interessato dal vincolo su entrambe le transazioni, in modo che la transazione che fa l'aggiornamento per prima blocchi l'altra, che quindi rileva un conflitto di scrittura.
In particolare, la prima transazione, che originariamente modificava il prezzo della giacca, accede in scrittura anche al documento dei pantaloni, aggiungendo un campo, *lock*, che valorizza a *true*. La seconda, che invece inizialmente modificava solo il prezzo dei pantaloni, accede in scrittura anche al documento della giacca, inserendo e valorizzando il nuovo campo *lock* con *true*. In questo modo, solo una delle due transazioni riesce a completare correttamente la scrittura, mantenendo il vincolo soddisfatto. L'altra, invece, va in abort perché rileva il conflitto.
In questo modo il contenuto del database resta coerente con i vincoli imposti.

Trattandosi di un artificio che serve solo per evitare l'anomalia ed essendo quindi privo di semantica, appena prima del termine della transazione il campo *lock* viene rimosso tramite l'operatore *unset*.

```
capiCollection.update_one({'nome': 'Pantalone'}, {'$set': {'lock': 'true'}}, session=session)
...
capiCollection.update_one({'nome': 'Pantalone'}, {'$unset': {'lock': 1}}, session=session)
session.commit_transaction()
```

Listing 4.28: Codice per la prevenzione di write skew sul terminale 1

```
capiCollection.update_one({'nome': 'Giacca'}, {'$set': {'lock': 'true'}}, session=session)
...
capiCollection.update_one({'nome': 'Giacca'}, {'$unset': {'lock': 1}}, session=session)
session.commit_transaction()
```

Listing 4.29: Codice per la prevenzione di write skew sul terminale 2

Conclusioni

La gestione delle transazioni è un elemento fondamentale per garantire la coerenza e l'affidabilità dei dati in qualsiasi sistema di gestione di database.

MongoDB, adattandosi alle esigenze e alle richieste del settore, ha introdotto le transazioni multi-documento a partire dalla versione 4.0, mentre le operazioni su singolo documento erano già atomiche, e continuano ad esserlo, per natura.

Una transazione garantisce consistenza e integrità dei dati, assicurando che la sequenza di operazioni al suo interno sia atomica.

La sperimentazione ha dimostrato che MongoDB ha una buona capacità di gestione delle anomalie generalmente considerate in letteratura, che altri DBMS, tra cui MySQL, non sono in grado di evitare in maniera assoluta.

Per come MongoDB implementa le transazioni, vengono garantiti un elevato livello di isolamento, affidabilità e consistenza, anche se si scelgono livelli di read e write concern inferiori.
Ad ogni modo, in applicazioni critiche, è sempre preferibile adottare livelli di isolamento superiori che garantiscono la massima coerenza; tuttavia, è da tenere in considerazione il fatto che questo comporta un maggior overhead, soprattutto in casi di sistemi che fanno un uso spinto della concorrenza, poiché tutte quelle transazioni in conflitto vengono bloccate e devono quindi essere rieseguite, rallentando complessivamente l'applicazione.
Una scelta consona del livello di isolamento nell'uso delle transazioni permette di rispondere alle esigenze di coerenza, senza sacrificare le prestazioni.