MongoDB:

Intro:

MongoDB è un sistema di gestione di database NoSQL orientato ai documenti, progettato per gestire grandi volumi di dati non strutturati o semi-strutturati. A differenza dei database relazionali tradizionali che utilizzano tabelle e righe, MongoDB utilizza documenti BSON (Binary json) e collezioni per memorizzare i dati.

Caratteristiche Principali di MongoDB

1. Orientato ai Documenti:
   - MongoDB memorizza i dati in documenti BSON, che sono simili ai documenti json. Questo permette di avere strutture dati flessibili e nidificate.
2. Scalabilità Orizzontale:
   - MongoDB supporta la shardizzazione, una tecnica che consente di distribuire i dati su più server per garantire alte prestazioni e scalabilità.
3. Alta Disponibilità:
   - Utilizzando Replica Set, MongoDB replica i dati su più server, garantendo tolleranza ai guasti e alta disponibilità.
4. Schema-less:
   - Non è necessario definire uno schema rigido per i dati, consentendo una grande flessibilità nell'aggiunta e modifica dei campi nei documenti.

Struttura MongoDB

1. Database:
   - Il database è il livello più alto dell'organizzazione dei dati in MongoDB.
   - Contiene una o più collezioni.
   - Se un database non esiste, MongoDB crea il database quando archivi per la prima volta i dati per quel database
   - Esempio di comando per selezionare un database: `use myNewDB`
2. Collezione:
   - Una collezzione è un gruppo di documenti all'interno di un database, analogo a una tabella nei database relazionali.
   - Le collezioni sono schemaless, il che significa che i documenti all'interno di una collezione possono avere campi diversi.
   - Esempio di comando per creare una collezione: `db.createCollection("myCollection")`
3. Documento:
   - Un Documento è l'unità base di dati in MongoDB, simile a una riga in una tabella relazionale.
   - Memorizzato in formato BSON (Binary json).
   - I documenti possono avere strutture nidificate e incorporare array e oggetti.
   - Esempio di documento:

```
{
        "_id": 1,
        "nome": "Mario Rossi",
        "età": 30,
        "indirizzo": {
```

```
                        "via": "Via Roma",
                        "numero": 42,
                        "città": "Milano"
                }
        }
```

Schema Design:

In MongoDB, ci sono due principali modalità per modellare relazioni tra i dati: *embedding* e *referencing*. Ognuna ha i suoi vantaggi e svantaggi, e la scelta dipende spesso dalle esigenze specifiche dell'applicazione.

Schema Embedded

L'embedding è utile quando si hanno dati strettamente correlati che vengono spesso letti insieme. In pratica, si inserisce un documento all'interno di un altro.

ESEMPIO:

Immaginiamo di avere un'applicazione che gestisce ordini di acquisto. Ogni ordine ha diversi prodotti. Con l'approccio embedded, ogni prodotto può essere memorizzato all'interno del documento dell'ordine.

Schema:

```
{
  "_id": ObjectId("60c5c5f3f1b22c1d2a5e5b78"),
  "customer": {
    "name": "Mario Rossi",
    "email": "mario.rossi@example.com"
  },
  "orderDate": "2023-05-27",
  "status": "processing",
  "items": [
    {
      "productId": ObjectId("60c5c5f3f1b22c1d2a5e5b79"),
      "productName": "Laptop",
      "quantity": 1,
      "price": 1000
    },
    {
      "productId": ObjectId("60c5c5f3f1b22c1d2a5e5b7a"),
      "productName": "Mouse",
      "quantity": 2,
      "price": 50
    }
  ]
}
```

In questo schema, ogni ordine contiene un array di prodotti (items). Questa struttura è efficiente per leggere e scrivere ordini completi, dato che tutte le informazioni sono contenute in un singolo documento.

Vantaggi dell'Embedding:

- Lettura più veloce quando i dati correlati sono spesso letti insieme.
- Meno query per ottenere dati completi.

- Coerenza dei dati garantita all'interno del documento.

Svantaggi dell'Embedding:

- Dimensione massima del documento di 16 MB.
- Difficoltà nella gestione di aggiornamenti frequenti in sub-documenti molto grandi.

Schema Referencing

Il referencing è utile quando si hanno dati correlati ma che possono crescere indefinitamente o che vengono letti/modificati separatamente. Invece di includere i documenti correlati direttamente, si memorizzano i riferimenti (gli ID) ai documenti correlati.

ESEMPIO:

Utilizziamo lo stesso esempio degli ordini e prodotti, ma questa volta utilizzeremo il referencing.

Schema Ordine:

```
{
  "_id": ObjectId("60c5c5f3f1b22c1d2a5e5b78"),
  "customer": {
    "name": "Mario Rossi",
    "email": "mario.rossi@example.com"
  },
  "orderDate": "2023-05-27",
  "status": "processing",
  "items": [
    ObjectId("60c5c5f3f1b22c1d2a5e5b79"),
    ObjectId("60c5c5f3f1b22c1d2a5e5b7a")
  ]
}
```

Schema Prodotto:

```
{
  "_id": ObjectId("60c5c5f3f1b22c1d2a5e5b79"),
  "productName": "Laptop",
  "price": 1000,
  "category": "Electronics"
}
```

```
{
  "_id": ObjectId("60c5c5f3f1b22c1d2a5e5b7a"),
  "productName": "Mouse",
  "price": 50,
  "category": "Electronics"
}
```

In questo schema, ogni ordine contiene solo i riferimenti (gli ID) ai prodotti. Per ottenere le informazioni dettagliate sui prodotti, sarà necessario eseguire query aggiuntive.

Vantaggi del Referencing:

- Flessibilità nella gestione di dati correlati che cambiano frequentemente.
- Adatto per relazioni molti-a-molti.
- Evita il problema della dimensione massima del documento.

Svantaggi del Referencing:

- Richiede più query per ottenere i dati correlati.
- Maggiore complessità nella gestione delle relazioni.

Quando Usare Embedded o Referencing

Usare Embedded quando:

- I dati correlati sono strettamente legati e vengono letti insieme frequentemente.
- Le dimensioni dei documenti rimangono gestibili (inferiori a 16 MB).
- I dati correlati non cambiano spesso.

Usare Referencing quando:

- I dati correlati sono grandi o crescono in modo indefinito.
- I dati correlati vengono modificati frequentemente e separatamente.
- È necessario mantenere la flessibilità nella gestione delle relazioni tra i dati.

La scelta tra embedding e referencing dipende quindi dalle esigenze specifiche dell'applicazione, dal tipo di accesso ai dati, dalle dimensioni dei documenti e dalla frequenza degli aggiornamenti.

Operazioni CRUD su MongoDB

Create

Le operazioni di creazione in MongoDB aggiungono nuovi documenti a una collezione.

1. `insertOne`: Inserisce un singolo documento nella collezione.

```
db.collection.insertOne({
  name: "Mario Rossi",
  age: 30,
  address: {
    street: "Via Roma",
    city: "Milano"
  }
});
```

2. `insertMany`: Inserisce più documenti nella collezione.

```
db.collection.insertMany([
  {
    name: "Luigi Bianchi",
    age: 25,
    address: {
      street: "Via Milano",
      city: "Roma"
    }
  },
  {
    name: "Anna Verdi",
    age: 40,
    address: {
      street: "Via Napoli",
```

```
      city: "Napoli"
    }
  }
]);
```

Read

Le operazioni di lettura in MongoDB recuperano documenti da una collezione.

1. **`find`**: Recupera uno o più documenti che corrispondono a un criterio di ricerca.

```
db.collection.find({ age: { $gt: 20 } });
```

2. **`findOne`**: Recupera un singolo documento che corrisponde a un criterio di ricerca.

```
 db.collection.findOne({ name: "Mario Rossi" });
```

3. **`countDocuments`**: Conta il numero di documenti che corrispondono a un criterio di ricerca.

```
db.collection.countDocuments({ age: { $gt: 20 } });
```

4. **`distinct`**: Restituisce i valori distinti di un campo specifico.

```
db.collection.distinct("city");
```

Update

Le operazioni di aggiornamento in MongoDB modificano i documenti esistenti in una collezione.

1. **`updateOne`**: Aggiorna un singolo documento che corrisponde a un criterio di ricerca.

```
db.collection.updateOne(
  { name: "Mario Rossi" },
  { $set: { age: 31 } }
);
```

2. **`updateMany`**: Aggiorna tutti i documenti che corrispondono a un criterio di ricerca.

```
db.collection.updateMany(
  { city: "Milano" },
  { $set: { city: "Rome" } }
);
```

3. **`replaceOne`**: Sostituisce un singolo documento che corrisponde a un criterio di ricerca con un nuovo documento.

```
db.collection.replaceOne(
  { name: "Mario Rossi" },
  {
    name: "Mario Rossi",
    age: 31,
    address: {
```

```
      street: "Via Roma",
      city: "Roma"
    }
  }
);
```

Delete

Le operazioni di cancellazione in MongoDB rimuovono documenti da una collezione.

1. **`deleteOne`**: Cancella un singolo documento che corrisponde a un criterio di ricerca.

```
db.collection.deleteOne({ name: "Mario Rossi" });
```

2. **`deleteMany`**: Cancella tutti i documenti che corrispondono a un criterio di ricerca.

```
db.collection.deleteMany({ age: { $lt: 25 } });
```

Pipeline e Aggregazione

La pipeline di aggregazione in MongoDB è una potente operazione che consente di eseguire trasformazioni e aggregazioni su un set di dati. Una pipeline è composta da più fasi, dove ogni fase trasforma i documenti e passa i risultati alla fase successiva.

Operazione di Pipeline	Descrizione	Esempio
`$match`	Filtra i documenti in ingresso per farli corrispondere a una condizione.	`{ $match: { status: "A" } }`
`$group`	Raggruppa i documenti in base a un campo specifico e può eseguire operazioni di aggregazione come somma, media, conteggio, ecc.	`{ $group: { _id: "$city", total: { $sum: "$amount" } } }`
`$project`	Modifica la struttura dei documenti, inclusi, esclusi o rinominando campi.	`{ $project: { name: 1, age: 1, city: 1 } }`
`$sort`	Ordina i documenti in base a un campo specifico.	`{ $sort: { age: -1 } }`
`$limit`	Limita il numero di documenti in uscita.	`{ $limit: 5 }`
`$skip`	Salta un numero specifico di documenti.	`{ $skip: 10 }`
`$unwind`	Decompone un array in più documenti, uno per ogni elemento dell'array.	`{ $unwind: "$items" }`
`$lookup`	Unisce documenti da un'altra collezione (join).	`{ $lookup: { from: "products", localField: "product_id", foreignField: "_id", as: "productDetails" } }`
`$out`	Scrive i risultati della pipeline in una nuova collezione.	`{ $out: "newCollection" }`
`$merge`	Unisce i risultati della pipeline in una collezione esistente.	`{ $merge: { into: "targetCollection", whenMatched: "merge", whenNotMatched: "insert" } }`
`$addFields`	Aggiunge nuovi campi ai documenti.	`{ $addFields: { totalCost: { $sum: ["$price", "$tax"] } } }`
`$set`	Sinonimo di `$addFields`, utilizzato per aggiungere o aggiornare campi.	`{ $set: { totalCost: { $sum: ["$price", "$tax"] } } }`

Operazione di Pipeline	Descrizione	Esempio
`$replaceRoot`	Sostituisce il documento corrente con un documento specificato.	`{ $replaceRoot: { newRoot: "$newDocument" } }`
`$count`	Conta il numero di documenti che attraversano la pipeline.	`{ $count: "totalDocuments" }`
`$facet`	Esegue operazioni di aggregazione multiple in parallelo e restituisce i risultati in un unico documento.	`{ $facet: { "categorizedByPrice": [ { $bucket: { groupBy: "$price", boundaries: [ 0, 100, 200, 300 ], default: "Other" } } ] } }`
`$bucket`	Raggruppa i documenti in base a un intervallo di valori specificato.	`{ $bucket: { groupBy: "$price", boundaries: [ 0, 100, 200 ], default: "Other", output: { "count": { $sum: 1 } } } }`
`$bucketAuto`	Raggruppa i documenti in un numero specifico di bucket con intervalli di valori distribuiti automaticamente.	`{ $bucketAuto: { groupBy: "$price", buckets: 4 } }`
`$sortByCount`	Raggruppa i documenti per un campo specificato e conta il numero di documenti in ogni gruppo, ordinandoli per conteggio.	`{ $sortByCount: "$category" }`
`$geoNear`	Effettua una ricerca geografica nei documenti.	`{ $geoNear: { near: { type: "Point", coordinates: [ -73.99279 , 40.719296 ] }, distanceField: "dist.calculated", maxDistance: 2, query: { category: "Parks" }, includeLocs: "dist.location", spherical: true } }`
`$redact`	Utilizzato per controllare l'accesso ai dati a livello di documento.	`{ $redact: { $cond: { if: { $gt: ["$level", 5] }, then: "$$DESCEND", else: "$$PRUNE" } } }`
`$sample`	Estrae un numero casuale di documenti dalla collezione.	`{ $sample: { size: 3 } }`
`$indexStats`	Restituisce statistiche sugli indici utilizzati nella collezione.	`{ $indexStats: {} }`
`$addToSet`	Aggiunge valori a un array solo se non sono già presenti.	`{ $group: { _id: "$_id", uniqueValues: { $addToSet: "$value" } } }`

Esempio di Pipeline Completa

Supponiamo di avere una collezione di ordini e vogliamo ottenere il totale delle vendite per ogni città, ordinato per totale in modo decrescente.

```
db.orders.aggregate([
  { $match: { status: "completed" } },
  { $unwind: "$items" },
  { $group: { _id: "$city", totalSales: { $sum: "$items.price" } } },
  { $sort: { totalSales: -1 } },
  { $project: { _id: 0, city: "$_id", totalSales: 1 } }
]);
```

In questa pipeline:

1. **`$match`** filtra gli ordini con stato "completed".
2. **`$unwind`** decompone l'array "items" in documenti individuali.
3. **`$group`** raggruppa i documenti per città e calcola il totale delle vendite.
4. **`$sort`** ordina i risultati in base al totale delle vendite in ordine decrescente.
5. **`$project`** ristruttura i documenti per includere solo la città e il totale delle vendite.

L'aggregazione di MongoDB è uno strumento potente che permette di eseguire analisi complesse sui dati direttamente all'interno del database, riducendo la necessità di elaborazione lato applicazione.

Ottimizzazione Delle Query

Struttura del Database Basata su un Caso Reale

Per analizzare le query in MongoDB, ho creato un esempio di database basato su un caso reale. L'esempio riguarda i contratti tra un'azienda intermediaria e varie officine. In questo contesto, possiamo individuare due entità principali:

1. Contratto Generale: L'entità che definisce il contratto tra l'officina e l'azienda intermediaria.
2. Allegato: Un'estensione del contratto generale che riguarda accordi tra l'officina e altre aziende terze. Queste aziende terze offrono all'officina la possibilità di trovare clienti tramite l'azienda intermediaria.

Schema della Struttura del Database

Per condurre l'analisi, ho creato due database con un design dello schema differente:

1. Schema Embedded:
   - In questo schema, i contratti generali contengono un array di allegati incorporati direttamente all'interno di ciascun contratto.
2. Schema Reference:
   - In questo schema, i contratti generali contengono solo un array di riferimenti a documenti di allegati che si trovano in un'altra collezione separata.

Questo setup consente di confrontare le prestazioni delle query tra i due approcci di modellazione dei dati: incorporazione (embedded) e referenziazione (reference).

Find Query

> Ho analizzato il caso in cui devo filtrare tutti i contratti a seconda della compagnia di con il quale hanno sottoscritto il contratto, per fare ciò utilizzo una pipeline di aggreate.

In questo caso l'influenza sulle performance, non c'è solo nel design del databse, ma anche nella costruzione della pipeline. In particolare si nota che alcuni stages posso essere èarticolarmente lenti, mentre in altri casi posso ottimizzare quelli già resennti.

Emebedded: In questo caso ho fatto dei test anche paragonando due pipeline differenti, una fatta con lo stage `$project` e una fatta con la stage `$group`. In questo modo si evidenziano le differenze di performance che ci sono tra le due:

```
~ >>> py find.query.py
→ Find with $group 502206 Embedded: 27.24188542366028 s
→ Find with $project 502206 Embedded: 6.312788009643555 s
```

Pipeline con `$project`:

```
contracts_embedded
        .contracts
        .aggregate([
                {
            "$unwind": {
              "path": "$attachments"
            }
                },
                {
            "$project": {
              "_id": "$attachments._id",
              "company": "$attachments.company",
              "repair_shop_name": "$repair_shop_name",
              "repair_shop_p_iva": "$repair_shop_p_iva"
            }
          },
         {
                        "$match": {
                                "company": "HERTZ"
                        }
                }
        ])
```

1. **`$unwind`**:
   - Decompone l'array `attachments`, creando un documento separato per ogni elemento dell'array.
2. **`$project`**:
   - Seleziona e rinomina i campi desiderati, mantenendo solo `_id`, `company`, `repair_shop_name` e `repair_shop_p_iva` dai `attachments`.
3. **`$match`**:
   - Filtra i documenti dove `company` è uguale a "HERTZ".

Pipeline con `$group`:

```
contracts_embedded
        .contracts
        .aggregate([
                {
            "$unwind": {
              "path": "$attachments"
            }
                },
                {
            "$group": {
              "_id": "$attachments._id",
              "company": { "$first": '$attachments.company' },
              "repair_shop_name": { "$first": '$repair_shop_name' },
              "repair_shop_p_iva": { "$first": '$repair_shop_p_iva' },
            }
```

```
        },
            {
                "$match": {
                    "company": "HERTZ"
                }
            }
    ])
```

1. **`$unwind`**:
   - Come sopra, decompone l'array `attachments`.
2. **`$group`**:
   - Raggruppa i documenti per `_id` di `attachments`, mantenendo il primo valore di `company`, `repair_shop_name` e `repair_shop_p_iva`.
3. **`$match`**:
   - Filtra i documenti dove `company` è uguale a "HERTZ".

Differenze Chiave

- **`$project`**: Seleziona e rinomina campi specifici senza modificare la struttura generale del flusso di dati.
- **`$group`**: Raggruppa i documenti in base a un campo specificato, utile per operazioni di aggregazione e deduplicazione.

Referencing: Anche in questo caso ho paragonato due differenti pipeline con una sottile differenza per ottimizzare il `$match`. Gli stages sono uguali a quelli di prima, ma sono preceduti da lo stage `$lookup`.

Lo stage `$lookup` in MongoDB è utilizzato per eseguire operazioni di join tra due raccolte, simile a una join in SQL. Permette di combinare i documenti da diverse raccolte in una singola pipeline di aggregazione.

Per ottimizzare la pipeline si può passare allo stage lookup una sottopipeline da eseguire nella collection con la quale farà la join. In questo caso infatti passo dalla prima forma dove la match è posta come ultimo stage della pipeline alla seconda dove faccio la match direttamente nella collection `contracts_attachments`

```
contracts_reference
        .contracts_general
        .aggregate([
          {
            "$lookup": {
              "from": "contracts_attachments",
              "localField": "attachments",
              "foreignField": "_id",
              "as": "attachments",
            }
          },
              {
            "$unwind": {
              "path": "$attachments"
            }
              },
              {
            "$group": {
              "_id": "$attachments._id",
              "company": { "$first": '$attachments.company' },
              "repair_shop_name": { "$first": '$repair_shop_name' },
              "repair_shop_p_iva": { "$first": '$repair_shop_p_iva' },
            }
```

```
        },
            {
                "$match": {
                    "company": "HERTZ"
                }
            }
    ])
```

```
contracts_reference
    .contracts_general
    .aggregate([
      {
        "$lookup": {
          "from": "contracts_attachments",
          "localField": "attachments",
          "foreignField": "_id",
          "as": "attachments",
          "pipeline": [
                                {
                                    "$match": {
                                        "company": "HERTZ"
                                    }
                                }
          ]
        }
      },
            {
        "$unwind": {
          "path": "$attachments"
        }
            },
            {
        "$group": {
          "_id": "$attachments._id",
          "company": { "$first": '$attachments.company' },
          "repair_shop_name": { "$first": '$repair_shop_name' },
          "repair_shop_p_iva": { "$first": '$repair_shop_p_iva' },
        }
      }
    ])
```

Le differenze di performance sono meno nette rispetto a prima, ma comunque visibili:

```
~ >>> py find.query.py
→ Find fast 502260 Referencing: 15.351619720458984 s
→ Find slow 502260 Referencing: 16.658145904541016 s
```

PARAGONE TRA REFERENCING E EMBEDDING

Le differenze di performace tra i due design sono ben visibili in una query di questo tipo, dove il design embedded ha la meglio.

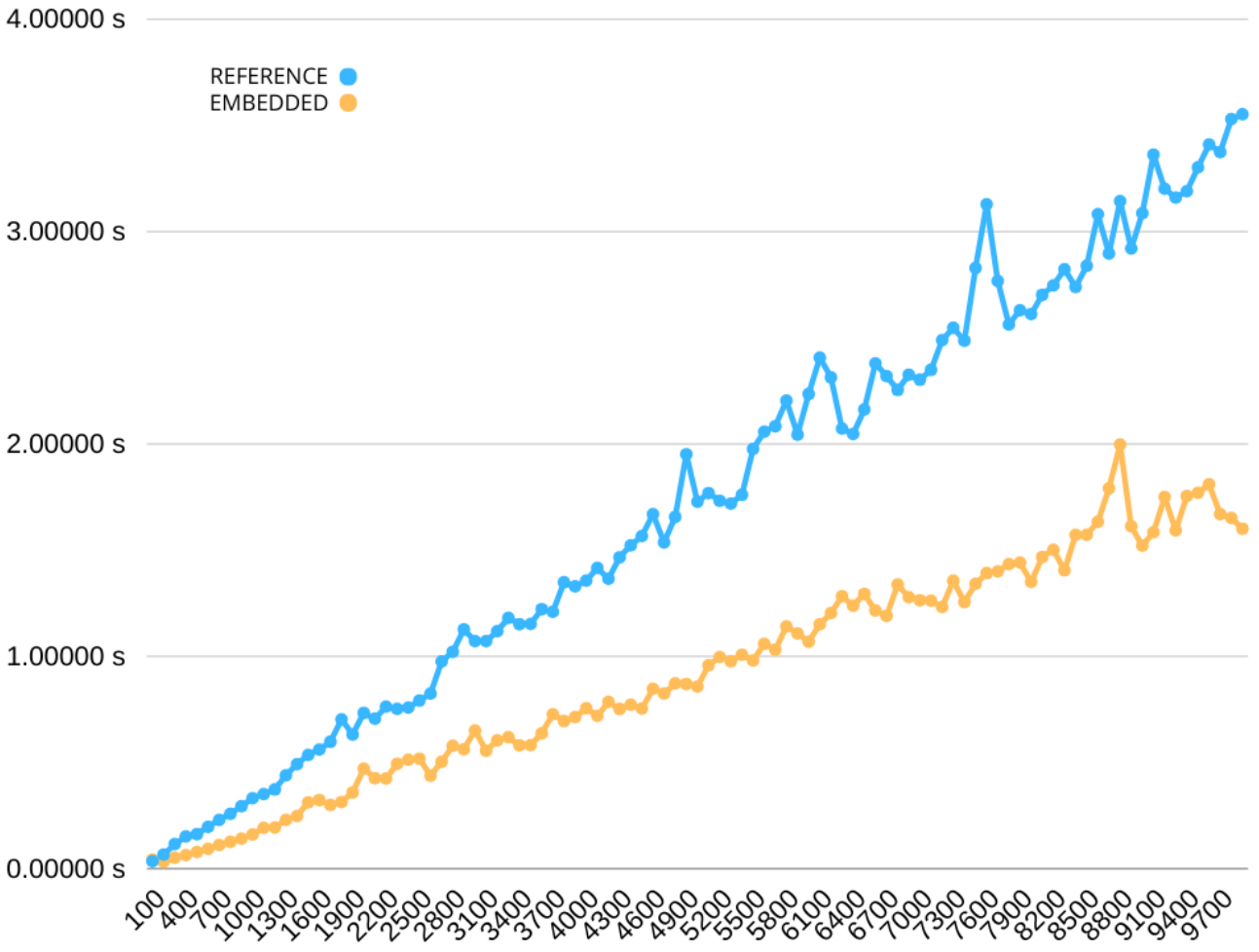
```
~ >>> py find.query.py
→ Find 502206 Embedded: 6.312788009643555 s
```

```
→ Find 502260 Referencing: 15.351619720458984 s
```

Create Query

> Analizzo le query create in due casi. Nel primo solo l'inserimento di un contratto generale e nel secondo analizzo l'inserimento solo dell'allegato.

Inserimento contratto generale: Analizzo le performance delle due strategie di design e come si sviluppano a seconda del numero di scritture. Si nota che la il design Reference con l'aumentare delle scritture aumenta si sposta dai tempi del design embedded.



Inserimento Allegato In questo caso analizzo le performance di un inserimento in tutti i campi del db di un allegato e confronto i tempi sui due modelli diffrenti:

```
~ >>> py create.query.py
→ Create Attachments 1045428 Embedded: 19.615631580352783 s
→ Create Attachments 1045395 Referencing: 10.929465293884277 s
```

In questo caso si nota un andamento contrario rispetto al caso prima. Proprio per la sua struttura a documenti annidati uno nell'altro, inserire tanti allegati in documenti diversi, è molto più lento. Mentre nel design reference creo per ogni nuovo allegato un nuovo documento nel databse degli allegati senza dover accedere in altri documenti annidati. Inserisco soltanto il rifermento all'interno dei documenti che rapprensentano i contratti generali.

Update Query

> Analizzo la query per modificare l'indirizzo di un autoriparatore all'interno dei contratti generali. In questo esempio il design non cambia i tempi di accesso al db che rimangono gli stessi

Design Embedding:

```
contracts_embedded
        .contracts
        .updateMany(
                { "repair_shop_street": "vicolo Lodi 48" },
                {
                        "$set": {
                                "repair_shop_street": "via Lodi 100"
                        }
                }
        )
```

Desing Referencing

```
contracts_reference
        .contracts_general
        .updateMany(
                { "repair_shop_street": "vicolo Lodi 48" },
                {
                        "$set": {
                                "repair_shop_street": "via Lodi 100"
                        }
                }
        )
```

Il tempo di accesso registrato è:

```
~ >>> py update.query.py
→ Update 1030429 Embedded: 10.251185178756714 s
→ Update 1030368 Referecing: 9.815329008102417 s
```

Delete Query

> Analizzo una query per elimanare una serie di allegati, filtrando per compagnia e citta

In questo caso le differenze delle query non si presentano solo nelle performance, ma anche nella complessità delle funzioni scritte per poter svolgere la stessa attività.

Performance:

```
~ >>> py delete.query.py
→ Delete 701 Attachments Embedded: 3.2612862586975098 s
→ Delete 725 Attachments Referencing: 35.2157278060913 s
```

per poter elimanare tutti i contratti all'interno del database con le collection con un design referencing, ho bisogno accedere sia all'array degli allegati e rimuovere gli id da li e anche rimuovere gli allegati interi dalla colleciton degli allegati.

Transaction:

Il controllo delle transazioni in MongoDB è una funzionalità che permette di garantire la consistenza dei dati durante le operazioni che coinvolgono più documenti o collezioni all'interno di un singolo database. Questa

funzionalità è stata introdotta a partire dalla versione 4.0 di MongoDB e supporta transazioni multi-documento su collezioni replicati.

Concetti Chiave delle Transazioni in MongoDB

1. Atomicità: Le transazioni in MongoDB sono atomiche, il che significa che tutte le operazioni all'interno di una transazione sono eseguite o nessuna. Non è possibile avere una situazione in cui alcune operazioni sono eseguite mentre altre no.
2. Coerenza: Durante una transazione, il database mantiene la coerenza dei dati. Tutti i lettori vedono una vista coerente del database, anche se i dati vengono modificati da altre transazioni.
3. Isolamento: Le transazioni in MongoDB sono isolate, il che significa che le operazioni all'interno di una transazione sono invisibili agli altri processi fino a quando la transazione non è completata (commit).
4. Durabilità: Le transazioni confermate (commit) sono persistenti e non possono essere annullate senza un intervento manuale.

Utilizzo delle Transazioni in MongoDB

Le transazioni in MongoDB possono essere utilizzate principalmente in due contesti:

- Transazioni su Documenti Singoli: Operazioni che coinvolgono più documenti all'interno di una singola collezione.
- Transazioni su Diverse Collezioni: Operazioni che coinvolgono più collezioni all'interno dello stesso database.

Esempio di Utilizzo

Ecco un esempio semplificato di come eseguire una transazione in MongoDB utilizzando PyMongo:

```
from pymongo import MongoClient
from pymongo.write_concern import WriteConcern
import random
from datetime import datetime

# Connessione al server MongoDB
client = MongoClient('localhost', 27017)

# Selezione del database
db = client['contracts_reference']

# Esecuzione di una transazione
with client.start_session() as session:
    # Avvia la transazione
    with session.start_transaction(write_concern=WriteConcern(w='majority')):
        allegati = db['contracts_attachments']
        generali = db['contracts_general']

        # Esempio di operazioni di aggiornamento all'interno della transazione
        generali.update_one({"_id": 1}, {"$push": {"attachments": }}, session=session)

        # Esempio di operazione di inserimento all'interno della transazione
        new_doc = {"_id": 3, "name": "New Document"}
        new_contract = {
                                "service": "VINSURANCE",
                                "company": "SOS",
                                "pricing_table": {
                                  "price_1": random.randint(1, 100),
                                  "price_2": random.randint(1, 100),
```

```
                                    "price_3": random.randint(1, 100)
                                        },
                                "date_signature": datetime.today(),
                                "date_created": datetime.today(),
                                        "date_expiration": datetime.today()
                            }
        allegati.insert_one(new_doc, session=session)

        # Commit della transazione
        session.commit_transaction()
```

Spiegazione Dettagliata

- Inizializzazione della Transazione: Si inizia una transazione avviando una sessione con `client.start_session()`.
- Operazioni all'interno della Transazione: Tutte le operazioni di lettura e scrittura eseguite con `session=session` sono parte della transazione.
- Commit della Transazione: Il metodo `session.commit_transaction()` conferma tutte le operazioni eseguite nella transazione.
- Rollback della Transazione: In caso di errori o eccezioni durante la transazione, è possibile eseguire un rollback utilizzando `session.abort_transaction()` per annullare tutte le modifiche non confermate.

Considerazioni

- Supporto di Replica Set: Le transazioni in MongoDB sono supportate solo su replica set. Non sono disponibili su singoli nodi autonomi (standalone).
- Performance: L'uso delle transazioni può influenzare le prestazioni, quindi è consigliabile utilizzarle solo quando è strettamente necessario garantire la coerenza dei dati tra diverse operazioni.

Le transazioni in MongoDB offrono un meccanismo potente per garantire la consistenza dei dati in operazioni complesse che coinvolgono più documenti o collezioni, migliorando così l'affidabilità delle applicazioni in cui la coerenza dei dati è critica.

Security:

In MongoDB, ci sono diversi modelli di sicurezza e pratiche che è possibile implementare per proteggere i dati e garantire l'accesso sicuro al database. Ecco alcuni dei principali modelli di sicurezza e tecniche utilizzabili:

1. Autenticazione e Autorizzazione

- Autenticazione: MongoDB supporta diversi metodi di autenticazione, inclusi utenti e ruoli interni di MongoDB

  Esempio di creazione di un utente con autenticazione interna di MongoDB:

```
use admin
db.createUser({
  user: "adminUser",
  pwd: "adminPassword",
  roles: [{ role: "userAdminAnyDatabase", db: "admin" }]
})
```

- Autorizzazione: MongoDB utilizza il modello di controllo degli accessi basato su ruoli (Role-Based Access Control, RBAC) per consentire o negare l'accesso a database, collezioni e operazioni specifiche.

  Esempio di assegnazione di ruoli a un utente:

```
use test
db.createUser({
  user: "testUser",
  pwd: "testPassword",
  roles: [{ role: "readWrite", db: "test" }]
})
```

2. Connessioni Sicure

- TLS/SSL: Utilizzare connessioni crittografate con TLS (Transport Layer Security) o SSL (Secure Sockets Layer) per proteggere i dati in transito tra l'applicazione e il database.

  Esempio di configurazione di MongoDB per utilizzare TLS:

```
net:
  tls:
    mode: requireTLS
    certificateKeyFile: /path/to/server.pem
```

3. Auditing

- Auditing: Abilitare e configurare l'auditing per monitorare e registrare le operazioni sul database, come autenticazioni, autorizzazioni e modifiche ai dati. Questo aiuta a tracciare l'accesso non autorizzato e a conformarsi ai requisiti di sicurezza e compliance.

  Esempio di configurazione di auditing:

```
auditLog:
  destination: file
  path: /var/log/mongodb/auditLog.json
  format: JSON
```

4. Firewall e Access Control

- Firewall: Configurare firewall per limitare l'accesso al server MongoDB solo ai client autorizzati.
- IP Whitelisting: Utilizzare IP Whitelisting per consentire l'accesso solo da indirizzi IP specifici.

5. Backup e Disaster Recovery

- Backup dei Dati: Implementare politiche regolari di backup dei dati per proteggere contro la perdita di dati accidentalmente cancellati o corrotti.

Risorse:

codice: https://github.com/mazzi-ni/elaborato_CBD