# Introduzione a MongoDB

Database SQL VS MongoDB
Vantaggi di MongoDB
Quando è consigliabile usare MongoDB
I principali design pattern
MongoDB Query Language (MQL)
Transazioni
ReplicaSet VS Cluster

**Andrea Di Tella**
*Senior Software Developer*
andrea@netframe.it

netframe
HYBRID CLOUD COLLABORATION

# INTRODUZIONE A MONGODB

**NoSQL** significa Not only SQL, che indica un ampliamento dello stesso con nuove funzionalità e nuovi sviluppi applicativi, a partire dalla eliminazione delle Relazioni.

**MongoDB** è un DBMS open source con la particolarità principale di gestire i dati in Documenti (**Documents**), rappresentati tramite JSON e organizzati in Collezioni (**Collections**).

Col termine Documento si intende un'entità generica che porta con sé informazioni e dati.

# DIFFERENZE TA SQL E NOSQL

La tabella seguente mette in relazione le nomenclature di un database **MySQL**, rispetto a un database **MongoDB**.

| MySQL | MongoDB |
|---|---|
| Table | Collection |
| Row | Document |
| Column | Field |
| Relationship | Link & Embedded Documents |

# ESEMPIO DI DOCUMENT

Il campo **_id** è un campo di default che rappresenta l'identificativo univoco di tipo **ObjectId** di ogni Document:
https://www.mongodb.com/docs/manual/reference/method/ObjectId/

Leggendo il Document possiamo notare il campo **vehicles** aggiunto come elementi (ovvero, degli **Embedded Documents**, in quanto non fanno parte di una Collection definita) dell'oggetto JSON relativo a tale campo. Questi Embedded Documents hanno delle proprietà in comune, altre invece sono esclusive per un veicolo o per l'altro.

Il campo **children** invece è formato da soli **ObjectId** relativi ad altri Documents, quindi è sufficiente indicare gli ObjectId dei Documents interessati.

```
{
    "_id": ObjectId(24f58ay8u51w),
    "name": "Mario",
    "surname": "Rossi",
    "sex": "M",
    "age": 47,
    "hobby": ["music","reading","sport"],
    "vehicles": [
        {
            "company":"DeLorean",
            "vehicle":"DMC-12",
            "year":1981,
            "max-speed":"88 mph",
            "notes":["Viaggia nel tempo"]
        },
        {
            "company":"Fiat",
            "vehicle":"Panda 4x4",
            "year":1986,
            "max-speed":"70 km/h",
            "colour":"verde acqua"
        }
    ],
    "children":[
        ObjectId(61h98ayd35d1),
        ObjectId(2rk01ay88da3)
    ]
}
```

# I PRINCIPALI VANTAGGI

**PERFORMANCE**

La struttura non relazionale dei dati di MongoDB richiede una **minore potenza di elaborazione** per la ricerca e il recupero dei dati rispetto a un database relazionale.

**FLESSIBILITA'**

MongoDB ha un'**architettura a schema dinamico** che funziona con dati e storage non strutturati. Poiché i dati sono memorizzati in documenti simili a JSON, gli schemi possono essere modificati in modo dinamico senza causare downtime.

**PARTIZIONAMENTO**

MongoDB offre scalabilità orizzontale tramite il partizionamento, che si occupa di dividere i dati da set di grandi dimensioni, distribuendoli su più server. Se un server non è in grado di gestire un carico elevato, i dati possono essere **divisi e distribuiti automaticamente** senza che la loro elaborazione venga interrotta.

# QUANDO USARE MONGODB

## ANALYTICS IN TEMPO REALE

MongoDB è una scelta indicata per l'integrazione e l'elaborazione dei **Big Data**: enormi quantità di dati eterogenei, che per le loro dimensioni non possono essere elaborate dai database relazionali tradizionali.
Poiché MongoDB è senza schema, è possibile effettuare l'accesso e la memorizzazione di vari tipi di dati all'istante. Il supporto integrato di MongoDB per il **partizionamento** consente inoltre di **scalare i dati** orizzontalmente su più server. Inoltre, offre la flessibilità necessaria per unire centinaia di origini di dati in un'unica vista per gli analytics in tempo reale e l'integrazione dei dati.

## GESTIONE DEI CONTENUTI

MongoDB è un'opzione eccellente per la **gestione dei contenuti** e la distribuzione di siti web di e-commerce, pubblicazioni online e sistemi web in generale. Il suo modello di dati flessibile consente di memorizzare facilmente diversi tipi di contenuti. Tutti i contenuti correlati possono essere memorizzati in un unico documento, il che facilita l'aggiunta di nuove funzionalità e attributi.

# I PRINCIPALI DESIGN PATTERN

## POLYMORPHIC PATTERN

**Utilizzato quando documenti nella stessa collection sono presenti più somiglianze che differenze**. Implica l'identificazione dei campi comuni nei documenti, in modo da implementare un set di query che verranno eseguite dalle applicazioni. Il monitoraggio delle query e l'utilizzo di campi specifici nei documenti è utile a identificare le classi e ottimizzare e il codice applicativo.

## BUCKET PATTERN

Utilizzato per **dati suddivisi in serie temporali** in cui vengono acquisiti come flusso in un periodo di tempo. In MongoDB è molto più efficiente inserire questi dati in una serie di documenti, ciascuno contenente i dati per un particolare intervallo di tempo, al posto di creare un documento globale. Ad esempio, è possibile suddividere il set di dati in intervalli di un'ora e inserire tutte le letture per quell'ora in un array in un singolo documento. Il documento stesso avrà orari di inizio e fine che indicano il periodo coperto da questo intervallo.

# I PRINCIPALI DESIGN PATTERN

## OUTLIER PATTERN

Utilizzato nei casi in cui alcuni **dati di documenti non rientrano nello schema standard**. Si tratta di un modello di schema progettato per situazioni tipiche dei social network con grandi influencer, vendite di libri, recensioni di film, ecc... Il **documento principale viene contrassegnato da un flag** che indica la presenza di ulteriori dati specifici in altri documenti di overflow, che fanno riferimento al primo documento tramite il campo _id. Il flag verrà utilizzato dal codice dell'applicazione per effettuare query aggiuntive per recuperare i dati dai documenti di overflow.

## COMPUTED PATTERN

Viene utilizzato quando i **dati devono essere elaborati frequentemente** oppure quando il modello di accesso ai dati prevede un'**intensa attività di lettura**. Questo modello prevede di eseguire i calcoli in background, aggiornando periodicamente il documento principale. Ciò fornisce un'**approssimazione valida dei campi** o dei documenti calcolati senza doverli generare continuamente per le singole query. Ciò può ridurre significativamente il carico sulla CPU evitando la ripetizione degli stessi calcoli.

# I PRINCIPALI DESIGN PATTERN

## SUBSET PATTERN

Viene utilizzato quando il **set di dati da elaborare supera le risorse disponibili** della macchina. Normalmente sono presenti documenti di grandi dimensioni, contenenti molte informazioni che non vengono utilizzate dalle applicazioni. Questo modello suggerisce di **dividere i dati utilizzati di frequente e quelli utilizzati più raramente in due collection separate**. Un tipico esempio potrebbe essere un'applicazione di e-commerce che mantiene le 10 recensioni più recenti di un prodotto nella collection principale e sposta tutte le recensioni più vecchie in una collection secondaria interrogata solamente in caso di necessità.

## EXTENDED REFERENCE PATTERN

Viene applicato quando sono **presenti molte entità logiche diverse**, ciascuna con la propria collection specifica, ma **con l'esigenza di riunire queste entità** per delle funzionalità specifiche. Un tipico scenario di e-commerce potrebbe avere collection separate per ordini, clienti e articoli. Questo può avere un impatto negativo sulle prestazioni quando vogliamo raccogliere insieme tutte le informazioni per un singolo ordine. La soluzione è **identificare i campi a cui si accede frequentemente e duplicarli all'interno del documento** dell'ordine. Questo modello compensa l'onere della duplicazione dei dati con una riduzione del numero di query per raggruppare insieme le informazioni necessarie alle applicazioni.

# I PRINCIPALI DESIGN PATTERN

## APPROXIMATION PATTERN

Questo pattern è utile negli scenari in cui sono necessari **calcoli dispendiosi** in termini di risorse (tempo, RAM, CPU) ma dove la **precisione del dato non è richiesta**. Un esempio specifico può essere un contatore di likes per immagini o post di un social network oppure un contatore di visualizzazioni di una pagina web, in cui non è necessario conoscere il conteggio esatto. In queste situazioni, l'utilizzo di questo modello può ridurre notevolmente il numero di scritture, aggiornando il contatore solo dopo ogni 100 o più visualizzazioni, invece che dopo ogni singola visualizzazione.

## PREALLOCATION PATTERN

Viene utilizzato negli scenari in cui è necessario creare una **struttura vuota iniziale che verrà popolata successivamente**. Un esempio di utilizzo potrebbe essere quello di un sistema di prenotazione che gestisce giorno per giorno una risorsa (sala riunioni, auto a noleggio, hotel, etc…), tenendo traccia se è libera o già prenotata. Una struttura bidimensionale di risorse (x) e giorni (y) rende molto più semplice e meno dispendioso, verificare le disponibilità tramite query specifiche.

# MONGODB QUERY LANGUAGE (MQL)

E’ il linguaggio usato per **elaborare e interagire con i dati** su database. Ha sintassi javascript e funziona in modo simile a SQL con operazioni **CRUD** per creare, leggere, aggiornare ed eliminare documenti.

Prima di lavorare con MQL è necessario selezionare il database sul quale compiere le operazioni:

```
use nomedatabase;
```

# ESEMPI DI QUERY - INSERT

```
// inserimento singolo
db.products.insertOne({ item: "card", qty: 15 })
db.products.insertOne({ item: "box",  qty: 2, color: "blue" })
db.products.insertOne({ item: "card", qty: 3, color: "yellow", "available": 1})

// inserimento con array
db.products.insertOne({
    item : "storage",
    qty : 50000,
    color : "purple",
    sizes : ["S", "M", "L", "XL"],
    certifications: [
        {
            description: "ISO9001",
            year: 2022
        },
        {
            description: "ISO14001",
            year: 2023
        }
    ]
});
```

# ESEMPI DI QUERY - INSERT

```
// inserimento massivo
db.products.insertMany([
        {item: "box", qty: 50, type: "desk" },
        {item: "card", qty: 90, type: "floor" },
        {item: "box", qty: 100 }
])

// inserimento referenziato
var catId1 = ObjectId();
db.category.insert({ _id: catId1, name: "Software" });
var catId2 = ObjectId();
db.category.insertOne({ _id: catId2, name: "Hardware" });
db.products.insertOne({ item: "box",  qty: 30, cat : catId1 })
db.products.insertOne({ item: "card", qty: 40, cat : catId2 })
```

# ESEMPI DI QUERY - FIND

```
// equals string
db.products.find({ item: "box"})

// like string
db.products.find({ item: /o/})

// greater than or equals
db.products.find({ item: "box", qty : { $gte : 10 }})

// greater than
db.products.find({ item: "box", qty : { $gt : 10 }})

// less than or equals
db.products.find({ item: "box", qty : { $lte : 10 }})

// less than
db.products.find({ item: "box", qty : { $lt : 10 }})
```

# ESEMPI DI QUERY - FIND

```
// not null
db.products.find({ color: {$ne : null} })

// null or missing field
db.products.find({ color: {$exists:true}})
db.products.find({ color: {$exists:false}})

// null or missing referenced document
db.products.find({ cat: {$exists:true}})
db.products.find({ cat: {$exists:false}})
```

# ESEMPI DI QUERY - FIND

```
// array values
db.products.find({ sizes:{ $in : ["L","XL"] }})

// embedded documents
db.products.find({certifications: {$elemMatch: {year : 2022} }})

// and
db.products.find({ $and : [
                        {certifications: {$elemMatch: {year : 2022} }},
                        {item: "storage" },
]})

// or
db.products.find({ $or : [
                        {certifications: {$elemMatch: {year : 2021} }},
                        {certifications: {$elemMatch: {year : 2024} }},
]})
```

# ESEMPI DI QUERY - UPDATE

```
// aggiornamento singolo
db.products.updateOne(
        {
                qty : { $gte : 10 }
        },
        {
                $set: { qty : 1000}
        }
)

// aggiornamento multiplo
db.products.updateMany(
        {
                qty : { $gte : 10 }
        },
        {
                $set: { qty : 1000, updated : 2}
        }
)
```

# ESEMPI DI QUERY - UPDATE

```
// aggiornamento multiplo con rimozione di campi
db.products.updateMany(
        {
                "qty" : { $gte : 10 }
        },
        {
                $unset: { color : "" }
        }
)
```

# ESEMPI DI QUERY - DELETE

```
// delete singolo
db.products.deleteOne(
        {
                qty : { $gte : 3 }
        }
)

// delete multiplo
db.products.deleteMany(
        {
                qty : { $gte : 10 }
        }
)
```

# TRANSAZIONI

Una transazione in un database, rappresenta una **sequenza di operazioni che avrà esito positivo solo se ogni operazione all'interno della transazione è stata eseguita correttamente**.

Le transazioni fino a poco tempo fa erano per lo più assenti nei database NoSQL.

Esistono però, applicazioni per le quali le transazioni sono richieste anche con database orientati ai documenti, ad esempio quando si **storicizzano dati** oppure si eseguono **aggiornamenti massivi**.

# TRANSAZIONI

Le transazioni possono essere eseguite solo su istanze MongoDB in esecuzione come parte di un cluster. Per avviare una transazione in MongoDB, è necessari prima di tutti avviare una sessione con il comando:

```
var session = db.getMongo().startSession()
```

Questo comando crea una variabile di sessione che memorizzerà l'oggetto della sessione. E' possibile avviare la transazione chiamando il metodo startTransaction() come segue:

```
session.startTransaction({
      "readConcern": { "level": "snapshot" },
      "writeConcern": { "w": "majority" }
})
```

Per confermare la transazione e salvare permanentemente nel database i dati elaborati e inseriti, è necessario eseguire il metodo commitTransaction() sull'oggetto sessione:

```
session.commitTransaction()
```

# TRANSAZIONI

Il metodo startTransaction() accetta due opzioni.

**readConcern**
Supponiamo che si avvii una transazione, ma dopo averlo fatto un altro utente aggiunge un documento alla collection. L'impostazione del livello **readConcern** consente di specificare quali dati la transazione deve leggere quando si avviano le operazioni. Impostarla su **snapshot** significa che la transazione leggerà uno snapshot di dati che è stato impegnato dalla maggior parte dei nodi nel cluster.

**writeConcern**
In questo caso l'opzione **w** richiede che il cluster riconosca quando le operazioni di scrittura della transazione sono state accettate su un numero specificato di nodi nel cluster. Invece di un numero, possiamo impostare ad esempio **majority**, che specifica che la transazione verrà considerata terminata con successo solo quando la maggioranza dei nodi, porta a buon fine l'operazione di scrittura.

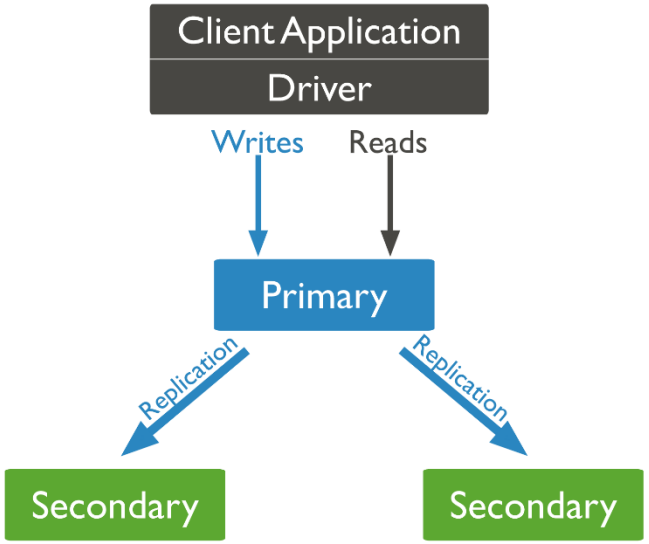
Questi valori di **readConcern** e **writeConcern** sono valori predefiniti sicuri da utilizzare nella maggior parte dei casi e forniscono garanzie affidabili di persistenza dei dati. Per approfondire altre casistiche si rimanda alla documentazione ufficiale di MongoDB:
https://www.mongodb.com/docs/manual/core/transactions/

# ReplicaSet VS Cluster

Ogni istanza di MongoDB che fa parte di un determinato replicaset è un membro del cluster. Ogni replicaset deve avere un membro primario e almeno un membro secondario.

**Il membro primario è il punto di accesso principale per le transazioni ed è anche l'unico membro che può accettare operazioni di scrittura**. Con la replicazione si copia innanzitutto l'oplog (registro delle operazioni) del primario. Successivamente, si ripetono le modifiche registrate sui rispettivi dataset dei secondari. Di conseguenza, ogni replica set può avere un solo membro primario alla volta.

Il meccanismo di failover automatico, in assenza di un primario, scatena un'elezione automatica tra i nodi secondari, scegliendo un nuovo primario.



# ReplicaSet VS Cluster

La principale differenze tra un replicaset e un cluster MongoDB è che il primo crea **diverse copie dello stesso insieme di dati** tra i nodi del replicaset, con lo scopo di offrire una soluzione di backup integrata e aumentare la disponibilità dei dati.

Un cluster invece, **distribuisce i dati su più nodi** attraverso una chiave shard (frammento). Questo processo frammenta i dati in tanti pezzi chiamati appunto shard. Successivamente, copia ogni shard su un nodo diverso. Un cluster ha lo scopo di supportare grandi insiemi di dati e operazioni ad alto rendimento, scalando orizzontalmente il carico di lavoro.

Di seguito il link alla documentazione ufficiale sui cluster in MongoDB:
https://www.mongodb.com/basics/clusters

# SLIDES PDF

Grazie!

https://www.netframe.it/mongodb

# netframe

HYBRID CLOUD COLLABORATION

Via Emilio Salgari 17
41123 Modena (MO)
P. IVA 04013790367

**www.netframe.it**