database NoSQL

Oltre i database relazionali

DBMG

Gestione dei dati al giorno d'oggi

Milioni di database relazionali utilizzati da applicazioni che funzionano molto bene, ma ...

Nuovi trend →

Organizzare dati non strutturati o semi-strutturati

Salvare dataset di grandi dimensioni offrendo scalabilità e prestazioni in modo economico

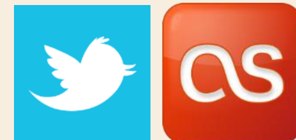
Reinventare i database relazionali

Nuove architetture

Sono emersi database non relazionali

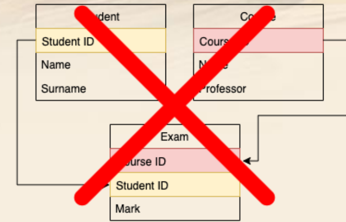
Nascita di “NoSQL”

- Nel **1998** Carlo Strozzi ha creato un database relazionare open-source, che richiedeva poche risorse, e che non utilizzava la classica interfaccia SQL
- Nel **2009** Johan Oskarsson’s (Last.fm) ha organizzato un evento per discutere i vantaggi dei database non-relazionali. È stato coniato un nuovo **hashtag** per promuovere l’evento su Twitter: **#NoSQL**



Principali caratteristiche di NoSQL

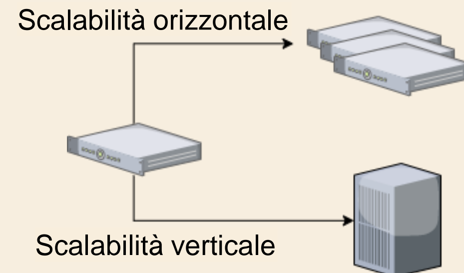
- **Nessuna operazione di join**



- **Senza Schema**
(nessuna tabella, schema implicito)

Student…	Name	Su…e
S123456	[illegible]	Rossi
S234567	Paolo	[illegible]

- **Scalabilità orizzontale**



Confronto

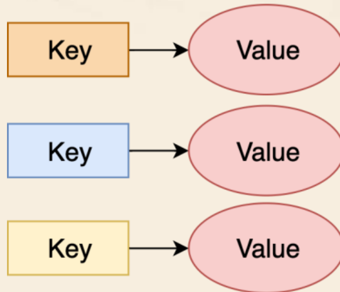
Database relazionale	Database non-relazionale
Basato su tabella, ogni record è una riga strutturata.	Soluzioni di archiviazione specializzate, ad esempio coppie di valori-chiave basate su documenti, database di grafi, archiviazione colonnare.
Schema predefinito per ogni tabella, modifiche consentite ma generalmente bloccanti (costose in ambienti distribuiti e live).	Senza schema: lo schema può cambiare dinamicamente per ogni documento, adatto per dati semi-strutturati o non strutturati.
Scalabile verticalmente, cioè tipicamente ridimensionato aumentando la potenza dell'hardware.	I database NoSQL sono scalabili orizzontalmente: vengono ridimensionati aumentando i server di database nel pool di risorse per ridurre il carico.
Utilizzo di SQL (Structured Query Language) per definire e manipolare i dati, un linguaggio molto versatile.	Linguaggi di query personalizzati, incentrati sul concetto di documenti, grafici e altre strutture di dati specializzate.

Confronto

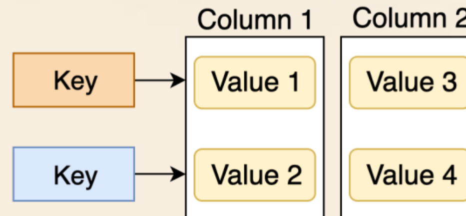
Database relazionale	Database non-relazionale
Adatto a query complesse, basato su join di dati.	Nessuna interfaccia standard per eseguire query complesse, nessun join.
Adatto per l'archiviazione di dati "piatta" e strutturata.	Adatto a dati complessi (ad esempio gerarchici), simili a JSON e XML.
Per esempio: MySql, Oracle, Sqlite, Postgres e Microsoft SQL Server.	**Per esempio:** MongoDB, BigTable, Redis, Cassandra, Hbase e CouchDB.

Tipi di database NoSQL

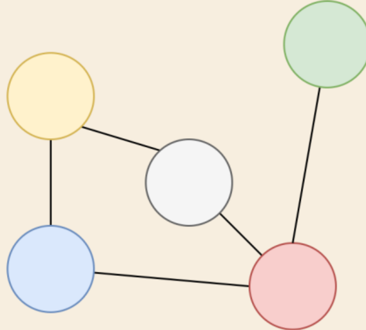
Chiave Valore



Orientati per Colonna



Database sui grafi

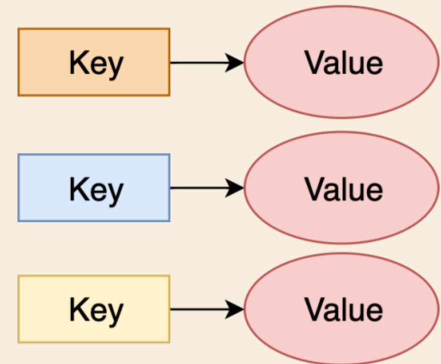


Basati sui documenti

```
{
first_name: "Mario",
last_name : "Rossi",
SSN: "AAAA00000000",
city: "Torino",
job: "Engineer",
cars: [
  {
    model: "Model S",
    year: "2018"
  },
  {
    model: "Model X",
    year: "2016"
  }
],
}
```

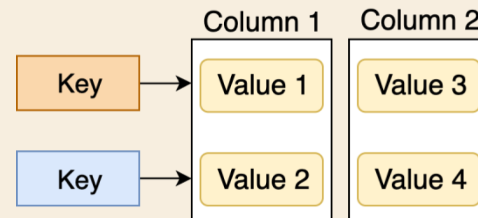
Database Chiave-valore

- Archivi di dati NoSQL **più semplici**
- Abbina le chiavi ai valori
- Nessuna struttura
- Ottime **performance**
- Scalabile facilmente
- Molto veloce
- Esempi: **DynamoDB**, Redis, Riak, Memcached



Database orientati per Colonna

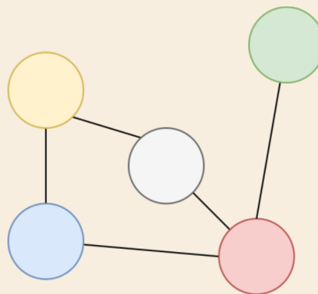
- Salvare i dati in un formato **colonnare**
  - Name = "*Mario*":row1,row3; "*Paolo*":row2,row4; …
  - Cognome = "*Rossi*":row1,row5; "*Bianchi*":row2,row6,row7…
- Una colonna è un **attributo** (anche complesso)
- Coppie chiave-valore archiviate e restituite sfruttando un sistema parallelo (simile agli **indici**)
- **Le righe** possono essere costruite da valori in colonna
- Le colonne salvate possono produrre in ouput dati semplici (**tabelle**)
- Esempi: **Cassandra**, Hbase, Hypertable



DBMG

Database sui grafi

- Basati sulla teoria dei grafi
- Creati con **Vertici** ed **Archi** orientati e non orientati tra coppie di vertici
- Utilizzati per salvare informazioni relativi alle **reti**
- Adatto per diverse applicazioni del mondo reale
- Esempi: Neo4J, Infinite Graph, OrientDB



DBMG

Database basati sui Documenti

- Memorizza e recupera documenti
- I documenti sono coppie auto descrittive (**attributo = valore**)
  - Come `city: "Torino"`
- Le chiavi sono mappate nei documenti
- I documenti hanno una natura **eterogenea**
- Tra le soluzioni più utilizzate
- Esempi: **MongoDB**, CouchDB, RavenDB

```
{
first_name: "Mario",
last_name : "Rossi",
SSN: "AAAA00000000",
city: "Torino",
job: "Engineer",
cars: [
  {
    model: "Model S",
    year: "2018"
  },
  {
    model: "Model X",
    year: "2016"
  }
],
}
```

Concetti utili: gli oggetti JSON

- JSON è un linguaggio indipendente per salvare e scambiare dati

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }  ],
}
```

Concetti utili: gli oggetti JSON

- JSON è un linguaggio indipendente per salvare e scambiare dati

Chiave

Valore

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }  ],
}
```

Concetti utili: gli oggetti JSON

- JSON è un linguaggio indipendente per salvare e scambiare dati

Chiave: "firstName" — Valore: "John"

Chiave: "address" — Valore composto: { "streetAddress" … "postalCode" }

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }  ],
}
```

Concetti utili: gli oggetti JSON

- JSON è un linguaggio indipendente per salvare e scambiare dati

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }  ],
}
```

Chiave: "firstName" — Valore: "John"

Chiave: "address" — Valore composto

Chiave: "phoneNumbers" — Vettore di valori

Database basati sui Documenti - MongoDB

Sistema NoSQL basato sui documenti

- Master-Slave / Server replicanti & Autosharding.
- Sistemi automatici per bilanciare il carico di lavoro grazie ai dati divisi in shard.
- Utilizza documenti codificati in formato JSON.
- Diversi tipi di indici come Indici B-tree, geospaziali.
- Memorizza documenti di qualsiasi dimensione senza complicare le applicazioni che lo utilizzano.

Database basati sui Documenti: caso d’uso

Caso d’uso reale:

- Applicazioni che richiedono la possibilità di archiviare attributi di tipo diverso e grandi quantità di dati.
- Applicazioni che richiedo un intensivo utilizzo dei dati con bassa latenza.

Aziende leader che utilizzano MongoDB

database NoSQL

Introduzione a MongoDB

DBMG

MongoDB: Introduzione

- MongoDB è il sistema di database più utilizzato tra quelli basate su documenti.
- Funzioni aggiuntive oltre alle standard di NoSQL:
  - Alte prestazioni
  - Disponibilità
  - Scalabilità nativa
  - Alta flessibilità
  - Open source

Terminologia – Concetti a confronto

Basi dati relazionali	Mongo DB
Tabella	Collezione
Record	Documento
Colonna	Campo

DBMG

MongoDB: design dei documenti

- Rappresentazione dei dati ad alto livello:
  - I record sono memorizzati sotto forma di documenti
    - Formati da coppie chiave-valore
    - Simili a oggetti JSON.
    - Possono essere nidificati.

```
{
_id: <ObjectID1>,
username: "123xyz",
contact: {
            phone: 1234567890,
            email: "xyz@email.com",
          }
access:   {
            level: 5,
            group: "dev",
          }
}
```

Embedded Sub-Document

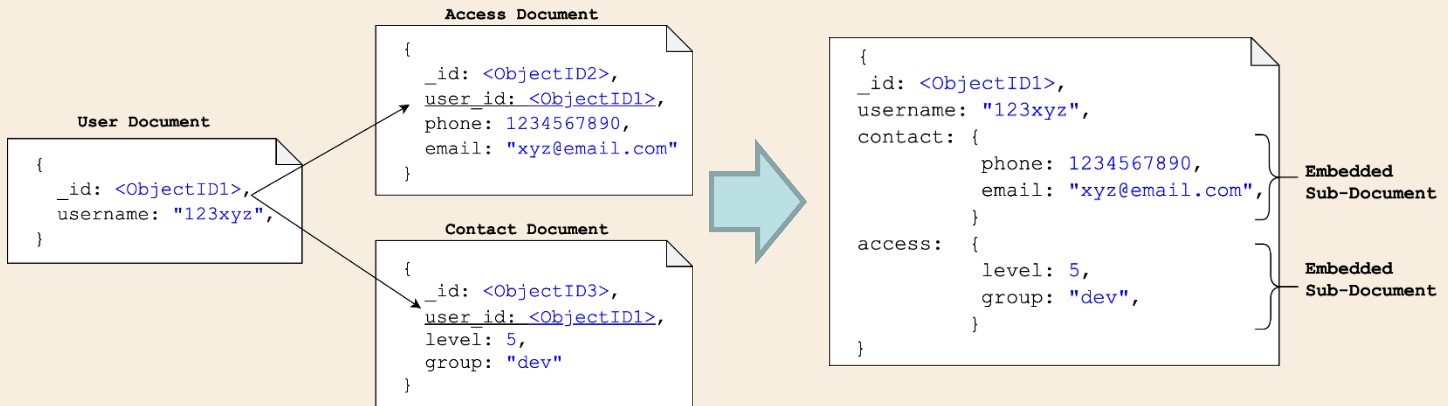
Embedded Sub-Document

MongoDB: design dei documenti

- Flessibile e con una ricca sintassi. Si adatta alla maggior parte dei casi d'uso.
- Permette il mapping dei tipi in oggetti dei principali linguaggi di programmazione:
  - anno, mese, giorno, timestamp,
  - liste, sotto-documenti, etc.

MongoDB: design dei documenti

- **Attenzione!**
  - Le relazioni tra documenti sono inefficienti.
    - Il riferimento viene fatto tramite l'uso dell'Object(ID). **Non** esiste l'operatore di **join nativo.**



MongoDB: Caratteristiche principali

- Linguaggio di query ricco di funzionalità:
  - I documenti possono essere creati, letti, aggiornati e cancellati.
  - Il linguaggio SQL non è supportato.
  - Sono disponibili delle interfacce di comunicazione per i principali linguaggi di programmazione:
    - JavaScript, PHP, Python, Java, C#, ..

Casi d’uso: MongoDB vs Oracle

- I casi d’uso più comuni di MongoDB includono:
  - Internet of Things, Mobile, Analisi Real-Time, Personalizzazione, Dati geo spaziali.
- Oracle è ritenuto più adatto per:
  - Applicazioni che richiedono molte transazioni complesse (ad esempio: un sistema di gestione di partite doppie).

From https://www.mongodb.com/compare/mongodb-oracle

Casi d'uso: MongoDB + Oracle

- I sistemi di prenotazione che gestiscono un sistema di prenotazione viaggi.
  - La parte principale del sistema di prenotazione dovrebbe utilizzare Oracle.
  - Quelle parti dell'applicazione che interagiscono con l'utente finale – pubblicano contenuti, si integrano ai social network, gestiscono le sessioni – sarebbe meglio gestirli con MongoDB.

From https://www.mongodb.com/compare/mongodb-oracle

MongoDB

Operatori per selezionare i dati

DBMG

MongoDB: query language

- La maggior parte delle operazioni disponibili in SQL può essere espressa nel linguaggio usato da MongoDB.

MySQL	MongoDB
`SELECT`	`find()`

SELECT * FROM people	db.people.**find()**

DBMG

MongoDB: operatore find()

MySQL	**MongoDB**
SELECT	find()

MySQL	MongoDB
SELECT **id**, **user_id**, **status** FROM people	db.people.find(**{ }**, { **user_id: 1**, **status: 1** })

MongoDB: operatore find()

MySQL	MongoDB
SELECT	find()

SELECT **id,**
       user_id,
       status
FROM people
	db.people.find(
	    { },
	    { **user_id: 1,**
	      status: 1
	    }
	)

Condizioni (WHERE)

Selezione (SELECT)

MongoDB: operatore find()

MySQL	MongoDB
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

```
SELECT *
FROM people
WHERE status = "A"
```

```
db.people.find(
    { status: "A" }
)
```

Condizioni (WHERE)

MongoDB: operatore find()

MySQL	MongoDB
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

SQL	MongoDB

`SELECT user_id, status`
`FROM people`
`WHERE status = "A"`
	`db.people.find(`
	`{ status: "A" },`
	`{ user_id: 1,`
	`status: 1,`
	`_id: 0`
	`}`
	`)`

Condizioni (WHERE)

Selezione (SELECT)

Di default, il campo `_id` viene sempre mostrato.
Per escludero dalla visualizzazione bisogna usare: `_id: 0`

MongoDB: operatori di confronto

- Nel linguaggio SQL, gli operatori di confronto sono essenziali per esprimere condizioni sui dati.
- Nel linguaggio usato da MongoDB sono disponibili con una sintassi differente.

MySQL	MongoDB	Descrizione
`>`	`$gt`	`Maggiore di`
`>=`	`$gte`	`Maggiore o uguale a`
`<`	`$lt`	`Minore di`
`<=`	`$lte`	`Minore o uguale a`
`=`	`$eq`	`Uguale a`
`<>`	`$neq`	`Diverso da`

MongoDB: operatori di confronto (>)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di

SELECT * FROM people WHERE **age > 25**	db.people.find({ **age: { $gt: 25 }** })

MongoDB: operatori di confronto (>=)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di
>=	**$gte**	**Maggiore o uguale a**

SQL	MongoDB
`SELECT * FROM people WHERE age >= 25`	`db.people.find( { age: { $gte: 25 } } )`

```
SELECT *
FROM people
WHERE age >= 25
```

```
db.people.find(
    { age: { $gte: 25 } }
)
```

MongoDB: operatori di confronto (<)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di
>=	$gte	Maggiore o uguale a
<	**$lt**	**Minore di**

SELECT * FROM people WHERE **age < 25**	db.people.find({ **age: { $lt: 25 }** })

MongoDB: operatori di confronto (<=)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di
>=	$gte	Maggiore o uguale a
<	$lt	Minore di
<=	**$lte**	**Minore o uguale a**

SQL	MongoDB

`SELECT *`
`FROM people`
`WHERE age <= 25`
	`db.people.find(`
	`{ age: { $lte: 25 } }`
	`)`

MongoDB: operatori di confronto (=)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di
>=	$gte	Maggiore o uguale a
<	$lt	Minore di
<=	$lte	Minore o uguale a
=	**$eq**	**Uguale a**

```
SELECT *
FROM people
WHERE age = 25
```

```
db.people.find(
    { age: { $eq: 25 } }
)
```

MongoDB: operatori di confronto (!=)

MySQL	MongoDB	Descrizione
>	$gt	Maggiore di
>=	$gte	Maggiore o uguale a
<	$lt	Minore di
<=	$lte	Minore o uguale a
=	$eq	Uguale a
<>	**$neq**	**Diverso da**

```
SELECT *
FROM people
WHERE age <> 25
```

```
db.people.find(
    { age: { $neq: 25 } }
)
```

MongoDB: operatori condizionali

- Per specificare condizioni multiple, **gli operatori condizionali** sono usati per affermare se una o entrambe le condizioni devono essere soddisfatte.
- Anche in questo caso MongoDB offre le stesse funzionalità di SQL con una sintassi diversa.

MySQL	MongoDB	Descrizione
AND	,	Entrambe soddisfatte
OR	$or	Almeno una soddisfatta

MongoDB: operatori condizionali (AND)

MySQL	MongoDB	Descrizione
AND	,	Entrambe soddisfatte

``SELECT *
FROM people
WHERE status = "A"
AND age = 50```
	``db.people.find(
	    { status: "A",
	      age: 50 }
	)```

MongoDB: operatori condizionali (OR)

MySQL	MongoDB	Descrizione
AND	,	Entrambe soddisfatte
OR	**$or**	**Almeno una soddisfatta**

```
SELECT *
FROM people
WHERE status = "A"
OR age = 50
```

```
db.people.find(
{ $or:
   [ { status: "A" } ,
     { age: 50 }
   ]
}
)
```

MongoDB: operatore count()

MySQL	MongoDB
COUNT	count() or find().count()

SELECT COUNT(*) FROM people	db.people.count() oppure db.people.find().count()

DBMG

MongoDB: operatore count()

MySQL	MongoDB
COUNT	count() or find().count()

- Analogamente all'operatore find(), count() può avere come argomento gli operatori condizionali.

SELECT COUNT(*) FROM people WHERE **age > 30**	db.people.count(**{ age: { $gt: 30 } }**)

MongoDB: ordinare i dati

- Per ordinare i dati rispetto a un attributo specifico bisogna utilizzare l'operatore sort().

MySQL	**MongoDB**
ORDER BY	sort()

SQL	MongoDB
SELECT * FROM people WHERE status = "A" **ORDER BY user_id ASC**	db.people.find({ status: "A" }).**sort({ user_id: 1 })**

MongoDB: ordinare i dati

- Per ordinare i dati rispetto a un attributo specifico bisogna utilizzare l'operatore sort().

MySQL	MongoDB
ORDER BY	sort()

SELECT * FROM people WHERE status = "A" **ORDER BY user_id ASC**	db.people.find({ status: "A" }).**sort({ user_id: 1 })**
SELECT * FROM people WHERE status = "A" **ORDER BY user_id DESC**	db.people.find({ status: "A" }).**sort({ user_id: -1 })**

MongoDB

Inserire, aggiornare e cancellare documenti

DBMG

MongoDB: inserire nuovi documenti

- Mongo DB permette di inserire nuovi documenti nella base dati. Ogni tupla SQL corrisponde a un documento in MongoDB.
- La chiave primaria `_id` viene automaticamente aggiunta se il campo `_id` non è specificato.

MySQL	MongoDB
`INSERT INTO`	`insertOne()`

MongoDB: inserire nuovi documenti

MySQL	MongoDB
INSERT INTO	insertOne()

MySQL	MongoDB
`INSERT INTO people(user_id, age, status) VALUES ("bcd001", 45, "A")`	`db.people.insertOne( { user_id: "bcd001", age: 45, status: "A" } )`

```sql
INSERT INTO
people(user_id,
       age,
       status)
VALUES ("bcd001",
        45,
        "A")
```

```javascript
db.people.insertOne(
   {
     user_id: "bcd001",
     age: 45,
     status: "A"
   }
)
```

MongoDB: inserire nuovi documenti

- In MongoDB è possibile inserire più documenti con un singolo comando usando l'operatore `insertMany()`.

```
db.products.insertMany( [
      { user_id: "abc123", age: 30, status: "A"},
      { user_id: "abc456", age: 40, status: "A"},
      { user_id: "abc789", age: 50, status: "B"}
   ] );
```

DBMG

MongoDB: aggiornare documenti esistenti

- I dati esistenti possono essere modificati a seconda delle necessità.
- Aggiornare le tuple richiede la loro selezione tramite delle condizioni di «WHERE»

MySQL	**MongoDB**

UPDATE <table>
SET <statement>
WHERE <condition>
	db.<table>.updateMany(
	   { <condition> },
	   { $set: {<statement>} }
	)

MongoDB: aggiornare documenti esistenti

MySQL	MongoDB
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>} })

UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" }})

MongoDB: aggiornare documenti esistenti

MySQL	**MongoDB**
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>} })

UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" }})
UPDATE people SET age = **age + 3** WHERE status = "A"	db.people.updateMany({ status: "A" } , { **$inc: { age: 3 }** })

L'operatore $inc incrementa il valore di un campo.

MongoDB: cancellare documenti

- Cancellare dati esistenti, in MongoDB corrisponde alla cancellazione del documento associato.
- In maniera simile a SQL, più documenti possono essere cancellati con un singolo comando.

MySQL	MongoDB
`DELETE FROM`	`deleteMany()`

DBMG

MongoDB: cancellare documenti

MySQL clause	MongoDB operator
DELETE FROM	deleteMany()

DELETE FROM people WHERE status = "D"	db.people.deleteMany({ status: "D" })

DBMG

MongoDB: cancellare documenti

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

`DELETE FROM people`
`WHERE status = "D"`
	`db.people.deleteMany(`
	`  { status: "D" }`
	`)`

`DELETE FROM people`	`db.people.deleteMany({})`

DBMG

MongoDB

Operatori di aggregazione

DBMG

Aggregazione su MongoDB

- Gli operatori di aggregazione processano i dati in input e ritornano il risultato delle operazioni applicate.
- I documenti entrano in una pipeline che consiste di più fasi che trasforma i documenti in risultati aggregati.

Aggregazione su MongoDB

Collection

```
db.orders.aggregate(
    $match phase ——► { $match: { status: "A" } },
    $group phase ——► { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```

orders:

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

$match →

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

$group →

Results

```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

Aggregazione su MongoDB: Group By

MySQL	MongoDB
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   }
] )
```

Aggregazione su MongoDB: Group By

MySQL	MongoDB
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   }
] )
```

Campo usato per l'aggregazione

Aggregazione su MongoDB: Group By

MySQL	MongoDB
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
         _id: "$status",
         total: { $sum: "$age" }
     }
   }
] )
```

Campo usato per l'aggregazione

Funzione di aggregazione

Aggregazione su MongoDB: Group By

MySQL	MongoDB
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Aggregazione su MongoDB: Group By

MySQL	MongoDB
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Fase di aggregazione: Specificare l'attributo e la funzione applicate durante il raggruppamento.

Aggregazione su MongoDB: Group By

SQL	MongoDB
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
   { $match: { total: { $gt: 1000
] )
```

Fase di aggregazione: Specificare l’attributo e la funzione applicate durante il raggruppamento.

Condizioni: specificare le condizioni come nel campo HAVING

MongoDB Compass

Interfaccia grafica per Mongo DB

DBMG

MongoDB Compass

- Consente di esplorare visivamente i dati.
- Disponibile per Linux, Mac, or Windows.
- Analizza i documenti e visualizza le strutture complesse all’interno delle collezioni
- Consente di visualizzare, comprendere e lavorare con i dati geo spaziali.

MongoDB Compass

MongoDB Compass - Connect

CREATE FREE ATLAS CLUSTER
Includes 512 MB of data storage.
Learn more

New Connection

Favorites

RECENTS

OCT 15, 2019 11:56 AM
bigdatadb.polito.it:27017

OCT 7, 2019 2:00 PM
bigdatadb.polito.it:27017

OCT 15, 2019 11:23 AM
bigdatadb.polito.it:27017

OCT 14, 2019 5:25 PM
bigdatadb.polito.it:27017

OCT 15, 2019 11:42 AM
bigdatadb.polito.it:27017

OCT 15, 2019 11:26 AM
bigdatadb.polito.it:27017

OCT 14, 2019 3:26 PM
bigdatadb.polito.it:27017

Connect to Host

Hostname: bigdatadb.polito.it
Port: 27017
SRV Record

Authentication: Username / Password
Username: Gestionali
Password: ••••••••••••••
Authentication Database: dbdmg

Replica Set Name:
Read Preference: Primary

SSL: Unvalidated (insecure)

SSH Tunnel: None

Favorite Name: e.g. Shared Dev, QA Box, PRODUCTION

- Si connette a un'istanza locale o remota di MongoDB.

MongoDB Compass

- Consente di avere una panoramica dei dati sotto forma di lista di documenti o tabella strutturata.

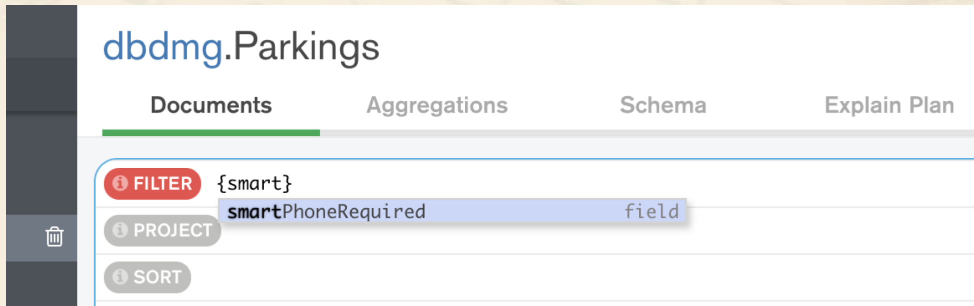
MongoDB Compass

- Analizza i documenti e i loro attributi.
- Supporta nativamente le coordinate geo spaziali.

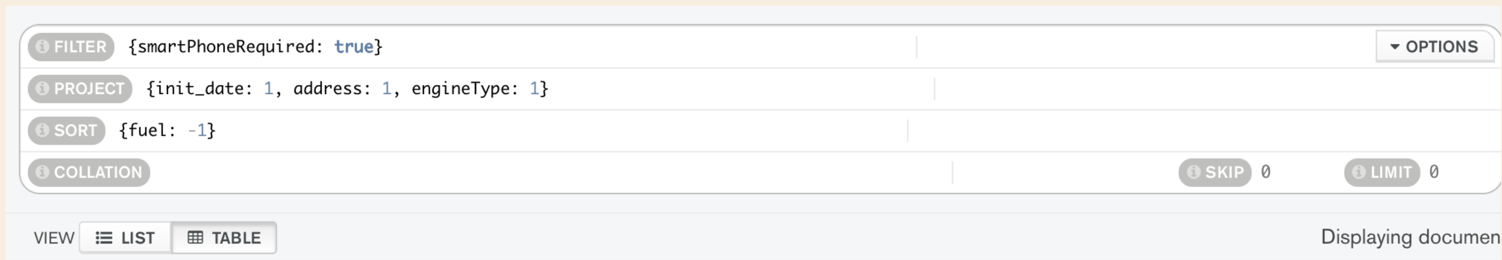
MongoDB Compass

- Consente di creare visivamente le interrogazioni ponendo delle condizioni sui dati.

MongoDB Compass



- Auto-completamento abilitato di default.



- Permette di costruire la interrogazioni passo passo.

DBMG

MongoDB Compass

- Analizza le performance di ogni interrogazione e fornisce suggerimenti per velocizzarla.

MongoDB Compass

MongoDB Compass - bigdatadb.polito.it:27017/dbdmg.Parkings

My Cluster
1 DBS 2 COLLECTIONS
filter
dbdmg
Bookings
Parkings

bigdatadb.polito.it:27017 STANDALONE MongoDB 3.6.14 Community

dbdmg.Parkings DOCUMENTS 100 TOTAL SIZE 48.4KB AVG. SIZE 496B INDEXES 5 TOTAL SIZE 55.9KB AVG. SIZE 11.2KB

Documents Aggregations Schema Explain Plan Indexes Validation

Validation Action ERROR Validation Level STRICT

```
{
  $jsonSchema: {
    required: ['exterior', 'interior', 'vendor', 'fuel'],
    properties: {
        vendor: {
          bsonType: "string",
          description: "must be a string"
        },
        fuel: {
          bsonType: "int",
          description: "must be an integer number"
        },
    }
  }
}
```

Validation modified CANCEL UPDATE

Sample Document That Passed Validation

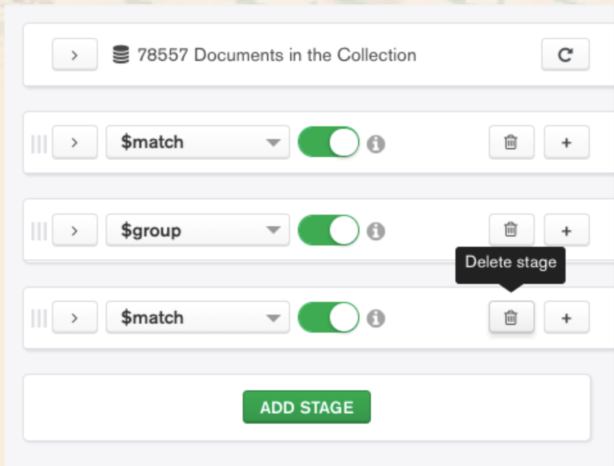
```
_id: ObjectId("59bef0cd2ad8532c2a60093d")
plate: 442
fuel: 37
vendor: "car2go"
final_time: 1505685847
loc: Object
init_time: 1505685697
vin: "VIN442"
smartPhoneRequired: true
```

Sample Document That Failed Validation

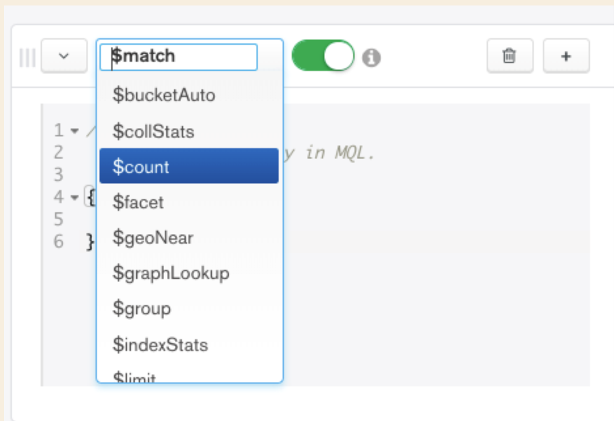
No Preview Documents

- Consente di specificare vincoli.
- Trova i documenti incompatibili.

MongoDB Compass: Aggregazione



- Consente di creare una a pipeline costituita da più fasi di aggregazione.



- Definisce dei filtri e degli attributi aggregati per ogni operatore.

MongoDB Compass: Fasi di aggregazione

$group

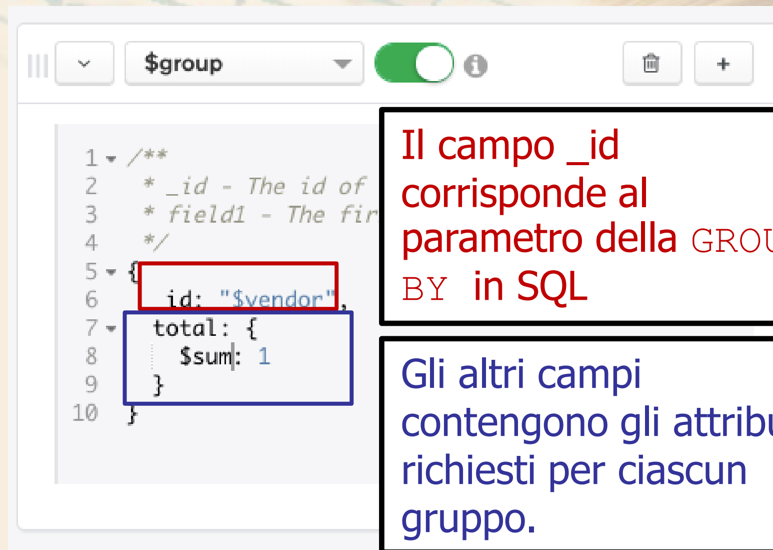
```
1  /**
2   * _id - The id of the group.
3   * field1 - The first field name.
4   */
5  {
6    _id: "$vendor",
7    total: {
8      $sum: 1
9    }
10 }
```

Output after $group stage (Sample of 2 documents)

```
_id:"car2go"
total:48423
```

```
_id:"enjoy"
total:30134
```

MongoDB Compass: Fasi di aggregazione



Il campo _id corrisponde al parametro della `GROUP BY` in SQL

Gli altri campi contengono gli attributi richiesti per ciascun gruppo.

Output after $group stage (Sample of 2 documents)

_id: "car2go" total: 48423	_id: "enjoy" total: 30134

Un gruppo per ciascun "vendor".

MongoDB Compass: Pipeline

$group

```
1  /**
2   * _id - The id of the group.
3   * field1 - The first field name.
4   */
5  {
6    _id: "$vendor",
7    total: { $sum: 1 },
8    avg_fuel : {$avg : "$fuel"}
9  }
10
```

Output after $group stage (Sample of 2 documents)

```
_id: "car2go"
total: 48423
avg_fuel: 64.88284492906264
```

```
_id: "enjoy"
total: 30134
avg_fuel: 61.03381562354815
```

Prima fase: raggruppamento per `vendor`

$match

```
1  /**
2   * query - The query in MQL.
3   */
4  {
5    avg_fuel: {$gt: 63},
6    total : {$gt : 35000}
7  }
```

Output after $match stage (Sample of 1 document)

```
_id: "car2go"
total: 48423
avg_fuel: 64.88284492906264
```

Seconda fase: condizione sui campi create nella fase precedente (`avg_fuel, total`).