# NoSQL databases

## Introduction to MongoDB (part 2)

# MongoDB

## Databases and collections.
## insert, update, and delete operations.

# MongoDB: Databases and Collections

- Each **instance** of MongoDB can manage multiple **databases**
- Each database is composed of a set of **collections**
- Each collection contains a set of **documents**
  - The documents of each collection represent **similar** "objects"
    - However, remember that MongoDB is **schema-less**
    - You are not required to define the schema of the documents a-priori and objects of the same collections can be characterized by different fields

# MongoDB: Databases and Collections

- Show the list of available databases
  - `show databases;`
- Select the database you are interested in
  - `use <database name>;`
- E.g.,

```
use deliverydb;
```

Note: shell commands vs GUI interface.

# MongoDB: Databases and Collections

- Create a database and a collection inside the database
  - Select the database by using the command
    `use <database name>`
  - Then, create a collection
    - MongoDB creates a collection implicitly when the collection is first referenced in a command
- Delete/Drop a database
  - Select the database by using use <database name>
  - Execute the command db.dropDatabase()
- E.g.,

```
use deliverydb;
db.dropDatabase();
```

# MongoDB: Databases and Collections

- A collection stores documents, uniquely identified by a document "**_id**"
- Create collections
  - `db.createCollection(<collection name>, <options>);`
  - The collection is associated with the current database. Always select the database before creating a collection.
  - Options related to the collection size and indexing, e.g., e.g., to create a capped collection, or to create a new collection that uses document validation
- E.g.,

```
db.createCollection("authors", {capped: true});
```

# MongoDB: Databases and Collections

- Show collections

```
show collections;
```

- Drop collections

```
db.<collection name>.drop();
```

- E.g.,

```
db.authors.drop();
```

# MongoDB: Read/Insert/Update data

| MongoDB | Relational database |
|---|---|
| `db.users.find()` | `SELECT * FROM users` |
| `db.users.insert({ user_id: 'bcd001', age: 45, status: 'A'})` | `INSERT INTO users (user_id, age, status) VALUES ('bcd001', 45, 'A')` |
| `db.users.update( { age: { $gt: 25 } }, { $set: { status: 'C'}}, { multi: true })` | `UPDATE users SET status = 'C' WHERE age > 25` |

# MongoDB: Insert documents

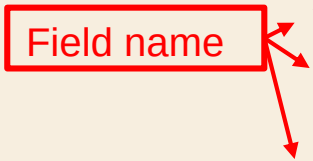
- Insert a single document in a collection
  - `db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );`
- E.g.,

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } );
```

# MongoDB: Insert documents

- Insert a single document in a collection
  - `db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );`
- E.g.,

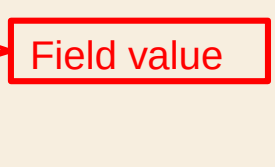
```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
} );
```



# MongoDB: Insert documents

- Insert a single document in a collection
  - db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );
- E.g.,

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
} );
```



# MongoDB: Insert documents

- Insert a single document in a collection
  - `db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );`

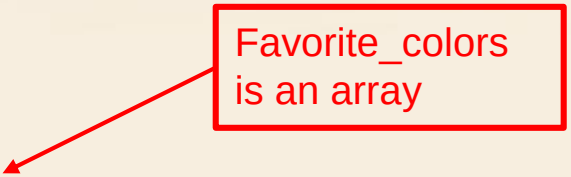
Now people contains a new document representing a user with:

```
user_id: "abc123",
age: 55
status: "A"
```

# MongoDB: Insert documents

- E.g.,

```
db.people.insertOne( {
    user_id: "abc124",
    age: 45,
    favorite_colors: ["blue", "green"]
} );
```




Now people contains a new document representing a user with:

`user_id: "abc124", age: 45`

and an array `favorite_colors` containing the values `"blue"` and `"green"`

# MongoDB: Insert documents

- E.g.,

```
db.people.insertOne( {
    user_id: "abc124",
    age: 45,
    address: {
        street: "my street",
        city: "my city"
    }
} );
```

Nested document

Example of a document containing a nested document

# MongoDB: inserting data

- New data needs to be **inserted into** the database.
  - Each SQL tuple corresponds to a MongoDB document
- The primary key `_id` is automatically added if the `_id` field is not specified.

| MySQL clause | MongoDB operator |
|---|---|
| `INSERT INTO` | `insertOne()` |

# MongoDB: inserting data

| MySQL clause | MongoDB operator |
|---|---|
| INSERT INTO | insertOne() |

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

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

# MongoDB: inserting data

- Insert multiple documents in a single statement: operator `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"}
   ] );
```

# MongoDB: Insert documents

- Insert many documents with one single command
  - `db.<collection name>.insertMany( [ <comma separated list of documents> ]);`
- E.g.,

```
db.people.insertMany([
{user_id: "abc123", age: 55, status: "A”},
{user_id: "abc124", age: 45,
  favorite_colors: ["blue", "green"]}
] );
```

# MongoDB: Document update

- Documents can be updated by using
  - `db.collection.updateOne(<filter>, <update>, <options>)`
  - `db.collection.updateMany(<filter>, <update>, <options>)`
  - `<filter>` = filter condition. It specifies which documents must be updated
  - `<update>` = specifies which fields must be updated and their new values
  - `<options>` = specific update options

# MongoDB: Document update

- E.g.,

```
db.inventory.updateMany(
    { "qty": { $lt: 50 } },
    {
      $set: { "size.uom": "in", status: "P" },
      $currentDate: { lastModified: true }
    }
)
```

  - This operation updates all documents with qty<50
  - It sets the value of the size.uom field to "in", the value of the status field to "P", and the value of the lastModified field to the current date.

# MongoDB: updating data

- Tuples to be updated should be selected using the WHERE statements

| MySQL clause | MongoDB operator |
|---|---|
| `UPDATE <table>`<br>`SET <statement>`<br>`WHERE <condition>` | `db.<table>.updateMany(`<br>`  { <condition> },`<br>`  { $set: {<statement>} }`<br>`)` |

# MongoDB: updating data

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

```
UPDATE people
SET status = "C"
WHERE age > 25
```

```
db.people.updateMany(
   { age: { $gt: 25 } },
   { $set: { status: "C" }}
)
```

# MongoDB: updating data

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

```
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 } }
)
```

The $inc operator increments a field by a specified value

# MongoDB: deleting data

- Delete existing data, in MongoDB corresponds to the deletion of the associated document.
  - Conditional delete
  - Multiple delete

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

# MongoDB: deleting data

| MySQL clause | MongoDB operator |
|---|---|
| DELETE FROM | deleteMany() |

| | |
|---|---|
| DELETE FROM people<br>WHERE status = "D" | db.people.deleteMany(<br>{ status: "D" }<br>) |

# MongoDB: deleting data

| MySQL clause | MongoDB operator |
|---|---|
| DELETE FROM | deleteMany() |

| | |
|---|---|
| DELETE FROM people<br>WHERE status = "D" | db.people.deleteMany(<br>{ status: "D" }<br>) |
| DELETE FROM people | db.people.deleteMany({}) |

# MongoDB

## Operational and design features

# MongoDB

## Transactions and sharding

# MongoDB: Main features

- MongoDB did not support **multi-document transactions**
  - **ACID** properties only at the **document level**

  You can use **embedded documents** and arrays to capture **relationships** between data in a single document structure instead of normalizing across multiple documents and collections

  Single-document atomicity obviates the need for multi-document transactions for many practical use cases.

- Since MongoDB 4.0, multi-document **transactions** are supported
  - Distributed transactions **across** operations, collections, databases, documents, shards
  - "Distributed Transactions" and "Multi-Document Transactions", starting in MongoDB 4.2, the two terms are synonymous.
  - This feature impacts on its **efficiency**

  In most cases, multi-document transaction incurs a greater **performance cost** over single document writes, and the availability of multi-document transactions should not be a replacement for **effective schema design**.

  For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases. That is, for many scenarios, **modeling your data appropriately will minimize the need for multi-document transactions**.

# MongoDB: Main features

- Horizontal scalability by means of **sharding**
  - Each shard contains a subset of the documents
  - Pay attention to the **sharding attribute**, as it impacts significantly on the performance of your queries
- *Horizontal Scaling* involves **dividing** the system dataset and load over **multiple servers**, adding additional servers to increase capacity as required.
  - While the overall speed or capacity of a single machine may not be high, each machine handles **a subset of the overall workload**, potentially providing better efficiency than a single high-speed high-capacity server.
  - Expanding the capacity of the deployment only requires **adding additional servers as needed**, which can be a **lower overall cost** than high-end hardware for a single machine.
  - The trade off is increased **complexity** in infrastructure and **maintenance** for the deployment.
- *Vertical Scaling* involves **increasing** the capacity of a **single server**, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
  - **Limitations** in available technology may restrict a single machine from being sufficiently **powerful** for a given workload.

https://docs.mongodb.com/manual/sharding/index.html

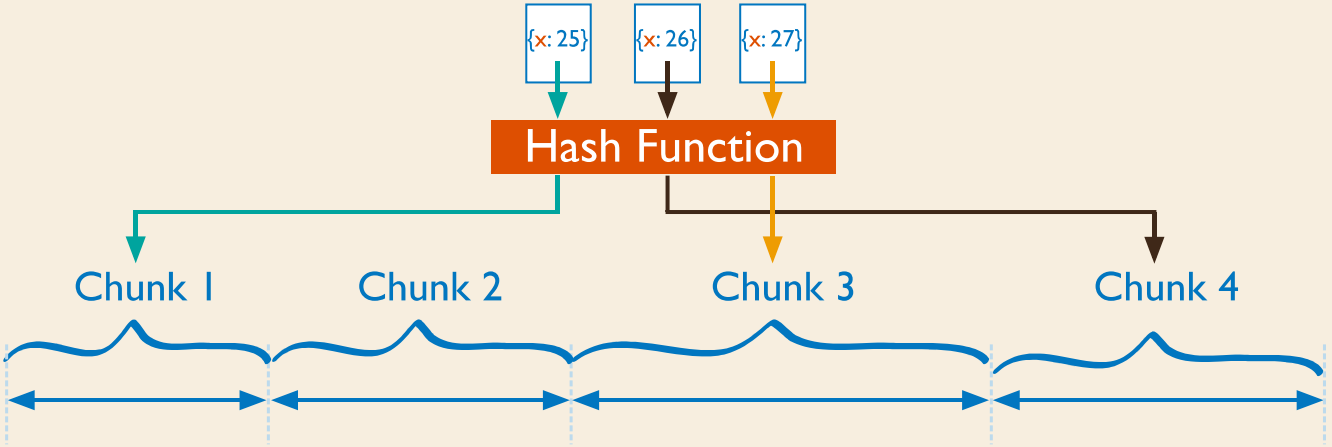
# MongoDB: Main features

- Horizontal scalability by means of **sharding**
  - Each shard contains a subset of the documents
  - Pay attention to the **sharding attribute**, as it impacts significantly on the performance of your queries
- MongoDB uses the **shard key** to distribute the **collection**'s documents across shards.
  - The shard key consists of **a field or fields** that exist in **every document** in the target collection. A sharded collection can have **only one** shard key.
  - The choice of shard key **cannot be changed** after sharding, nor can you unshard a sharded collection.
  - Although you cannot change which field or fields act as the shard key, starting in MongoDB 4.2, you can update a document's **shard key value** (apart from the _id field). Before MongoDB 4.2, a document's shard key field value is **immutable**.
  - To shard a non-empty collection, the collection must have an **index** that starts with the shard key.
  - The choice of shard key affects the **performance**, efficiency, scalability, and also the **availability** (HA) of a sharded cluster.
  - MongoDB distributes the **read and write** workload across the shards in the sharded cluster, allowing each shard to process a subset of cluster operations.

https://docs.mongodb.com/manual/sharding/index.html

# MongoDB: Main features

- Horizontal scalability by means of **sharding**
- MongoDB uses the **shard key** to distribute the **collection**'s documents across shards.




https://docs.mongodb.com/manual/sharding/index.html

# MongoDB: Main features

- Horizontal scalability by means of **sharding**
- MongoDB uses the **shard key** to distribute the **collection**'s documents across shards.