FUNDAMENTALS OF INFORMATION SYSTEMS PROJECT:

MongoDB

Letisia Ornelle Nguikoo
VR496684

SUMMARY

- Constructs of the data model
- Query language
- Distributed architecture
- Consistency model
- Conceptual model
- Physical data model

Constructs of the data model

What's MongoDB ?

NoSQL database document-oriented.

Stores information in documents.

Uses a BSON (Binary JSON) format to store data,

- Supports lower parse overhead than JSON.

No structural limit on data memorization.

Flexible schema:

- Not all documents in a collection are required to have the same fields.

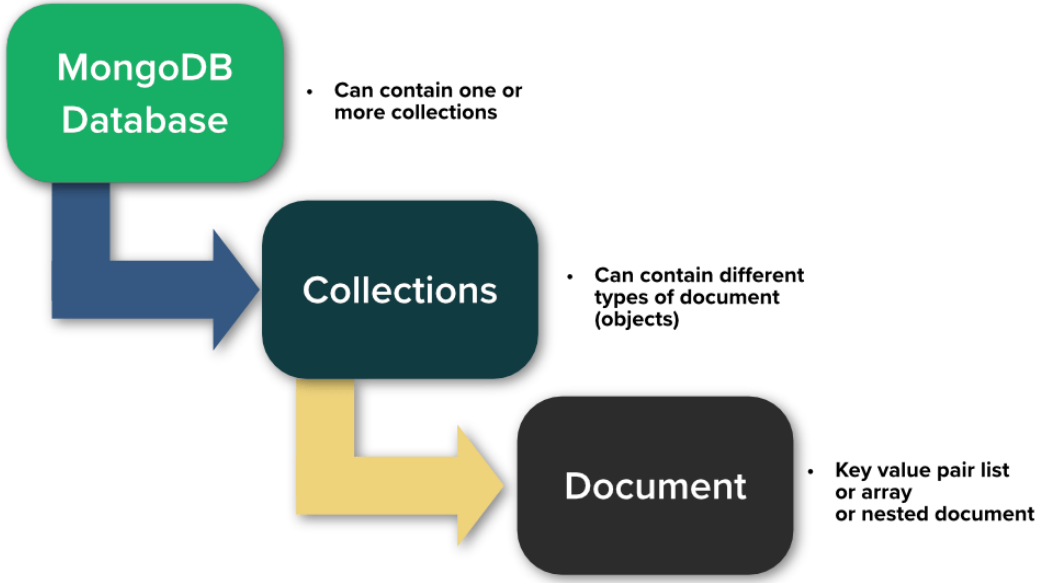
Easily adaptation:

- It's possible to add, remove o modify fields without having to make changes to the collection schema.

Constructs of the data model

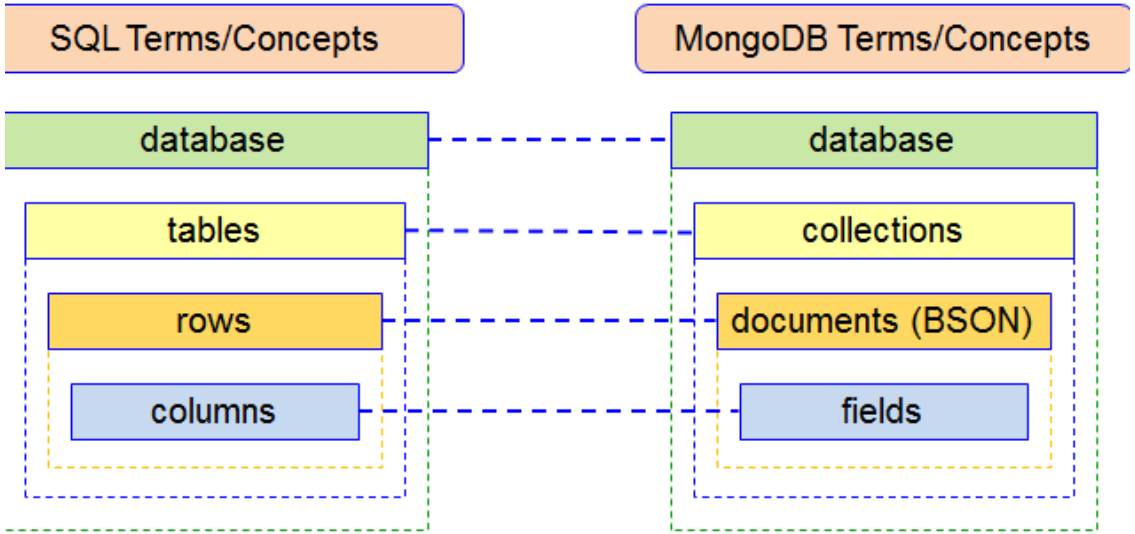
Components

- DATABASES
  - Provide a container to mantain and organize data.
- COLLECTIONS
  - A collection can’t extend on more databases.
  - Group documents with similar contents.
- DOCUMENTS
  - Each document rappresent an indipendent record.
  - Each document is a BSON entity and is associate to a unique identifier *_id*.



Constructs of the data model

Relational vs MongoDB



- MongoDB do not implement referential integrity constraint.

Contructs of the data model

Schema validation

- Allows the creation of validation rules for the fields, such as **allowed data types** and **value ranges.**
- Useful for ensuring data consistency and integrity:
  - Avoids the insertion of documents that contains errors or invalid data on the database.
  - Specifies rules to be respected by documents to be insert or update in a collection.
- Helps to ensure there are no unintended schema changes.

Query Language

MongoDB Query Language (MQL)

MQL is based on JavaScript.

Allows the execution of CRUD operations (Create Read Update Delete) on a database.

Provide powerful aggregation functions, operators to filter, sort, group and arrange data.

Query Language

Commands

- The most commonly used commands in MongoDB are:
  - Find (db.collection_name.find{})
  - Aggregate (db.collection_name.aggregate{})
- Find() selects documents in a collection and returns a cursor to the selected documents. It is similar to a SELECT SQL query in a relational database.
- Aggregate() operations process multiple documents and return computed results. We can use them to:
  - Group values from multiple documents together.
  - Perform operations on grouped data to return a single result.
  - Analyze data changes over time.

Query Language

Logical and comparison query operators

Logical operators return data based on expressions that evaluate to true or false.

Comparison operators return data based on value comparisons.

$and	Returns all documents that match the conditions of both clauses
$nor	Returns all documents that fail to match both clauses
$or	Returns all documents that match the conditions of either clause
$not	Returns documents that do not match the query expression
$gt-$gte	Matches values that are greater (or equal) than a specified value
$lt-$lte	Matches values that are less (or equal) than a specified value
$eq	Matches values that are equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values speficied in an array

Query Language

Aggregation operators

- MongoDB also offer aggregation operators.

$match	Filters the documents that match the specified condition(s)
$group	Separates documents into groups according to a «group key» and applies and aggregate function to each group.
$project	Used to fetch only the required fields of a document from a database
$sort	Sorts the resulting documents the way we require (ascending or descending)
$limit	Limits the number of returned documents by the query
$lookup	Can be used to provide a left outer join between two collections

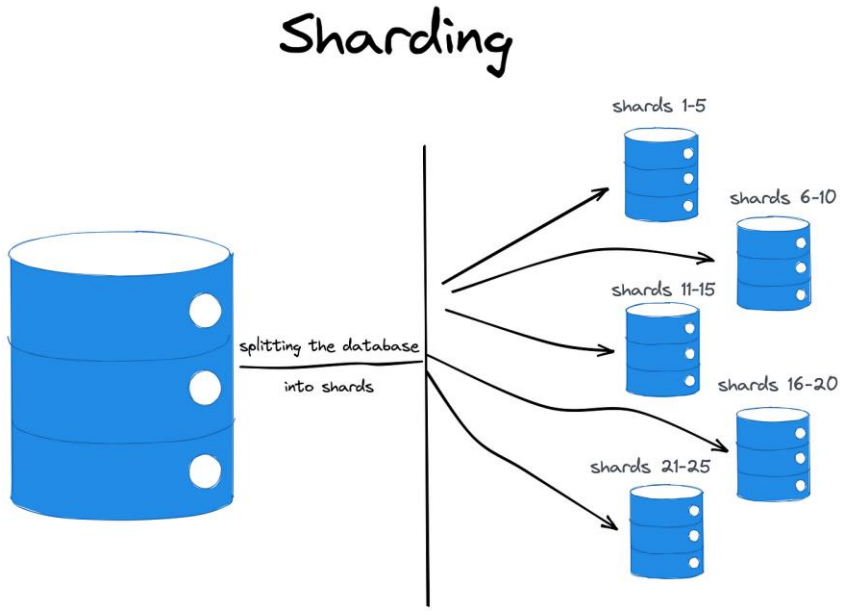
Distributed Architecture

- MongoDB supports a distributed architecture through the concept of **replication** and **sharding**, which allows to distribute data across multiple nodes to improve system performance, availability and scalability.

Distributed Architecture

Sharding

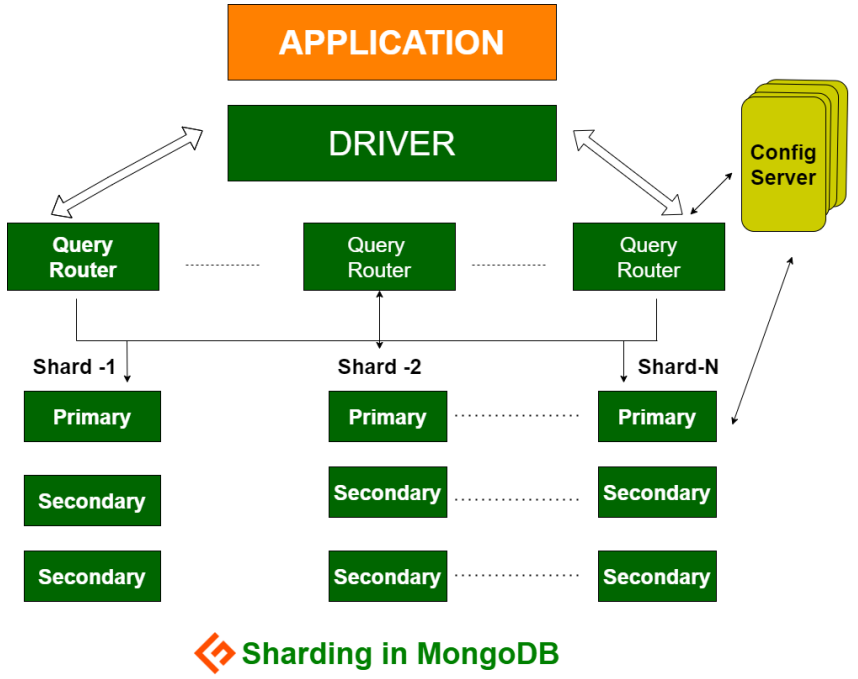
- Sharding is a method of **distribuiting** or **partitioning data** across multiple servers providing scalability and high availability.
- Data is divided into smaller chunks called **shards,**
  - Each shard is hosted on a separate server.
- It’s used when the size of the data exceeds the storage or processing capacity of a single server, by allowing to **scale horizontally.**
- A sharded cluster consists of:
  - Shard
  - Router(Mongos)
  - Config servers



Distributed Architecture

Sharding - Components

- **Shards**
  - Contain a subset of the cluster's data,
  - Can be added or removed to manage workload growth.
- **Router** (**mongos**) acts as an access point for client applications. Routers direct read and write operations to the appropriate shard nodes based on sharding rules and sharding keys.
- **Configuration server**: they maintain sharding metadata, store information about the distribution of data across various shards



Distributed Architecture

Data distribution

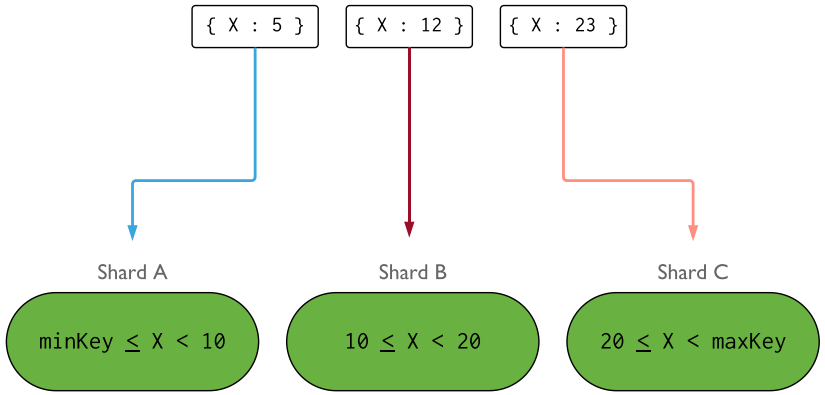
- In MongoDB, the **shard key** determines how the data is distributed across multiple servers
  - It’s a single or multiple indexed attribute in the documents that is choose to use for data distribution.
  - It’s use to decide to which shard or server the document should be stored.
- MongoDB uses two strategies to distribute documents across shard:
  - **Ranged sharding**
  - **Hashed sharding**

Distributed Architecture

Ranged sharding

- It is used to distribute data based on a range of shard key values.
- Each shard is associated to a specific value range of shard-key.
- Pro:
  - Documents with “close” shard key values are likely to be in the same shard.
  - It enables efficient queries where documents are read within a continuous range.
- Cons:
  - Read and write performance may decrease with poor shard key selection.

Example: Illustration of a sharded cluster using the field X as the shard key.

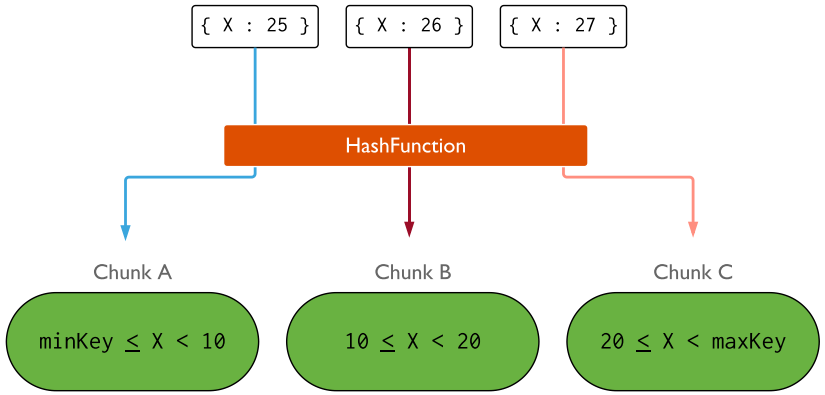


Distributed Architecture

Hashed sharding

- A hash function is applied to the shard key value, which generates a hash value,
  - The hash value is used to determine the shard where the document should be stored.
- Pro
  - It evenly distributes the data across shards, regardless of the distribution of the shard key values.
  - Can be useful when there's not a natural order or range in the data.
- Cons
  - Can make range-based queries more difficult.
  - Related data may be distributed across different nodes even though the shard key values may be close.

Example: Illustration of a sharded cluster using a hashed index on X.



Distributed Architecture

Sharding - Rebalance

- During the system life the workload can change and the initial sharding policy might be no better, thus unbalancing the cluster.
- MongoDB periodically assesses the balance of shards and performs rebalance operations, if needed.
- The unit of rebalance is called "**shard chunk**",
  - Each chunk contains contiguous values of shard keys.
- A background process known as the "**balancer**" automatically migrates chunks across the shards to ensure that each shard always has the same number of chunks.

Distributed Architecture

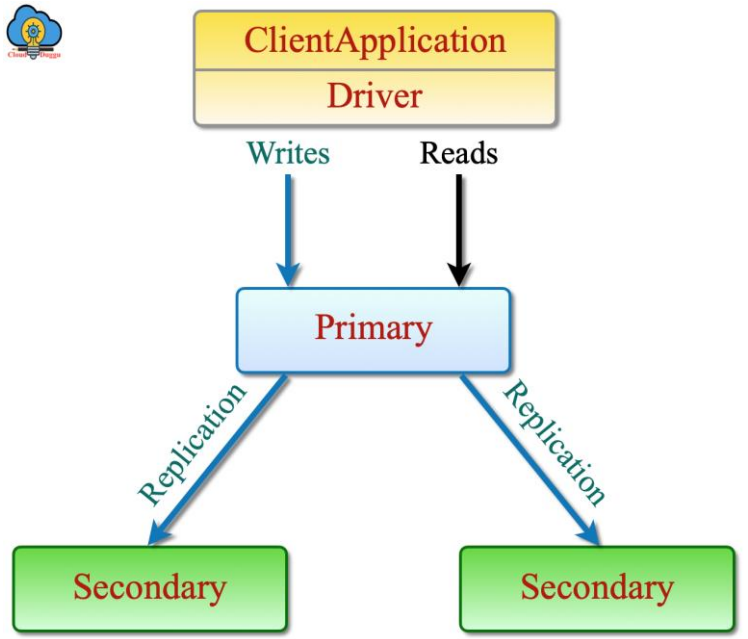
Replication

- Replication is provided through a feature called **replica set.**
- Allows to create copies of the data across multiple servers or nodes.
- Provides **redundancy**, **availability** and **fault tolerance.**
- Even if the replica set fails, the system can continue to operate smoothly without any downtime or loss of data.

Distributed Architecture

Replication – Replica Set

- A **replica set** is a group of MongoDB instances that maintain the same data.
- It consists of multiple nodes:
  - One primary node which receives writes operations.
  - Two or more secondary nodes which receive replicates data from the primary node.
- By default, the primary node receives write operations and replicates the data to the secondary nodes
- If the primary node fails, one of the secondary nodes automatically becomes the new primary.



Distributed Architecture

Replication – Replica Set

- The primary node stores information about document changes in the **oplog** collection.
- Members of a replica set communicate frequently via **heartbeat** messages.
- If a primary finds that it is no more able to receive **hearbeat** messages from more than half of the secondaries, it will renounce to its primary status and a new election will be called.
- The client can issue a **blocking call**, which will wait until the write has been received by all secondaries, or certain number of them.
- By default, clients read from the primary, however, they can specify a read preference to send read operations to a secondary.

Consistency Model

Strict consistency

- MongoDB is the system closest to relational, so it tries to guarantee the consistency as high as possible.
- By default, it ensures **strict consistency** on a single document.
- Users always see the system in the same state.
- Every read request returns the most updated value.
- When a write occurs, until the data is replicated on the other servers the next reading operations are blocked.

Consistency Model

Eventual consistency

- MongoDB becomes an eventually consistent system when the operations are done on the secondary members,
  - This happens because of a **replication lag**(delay between when the data is written to the primary and when it's available on the secondaries).
- Ensures that updates made to databases will **eventually** be reflected across all nodes,
  - Identical database queries will return the same results after some period of time.
- There may be a delay before all nodes have the same view of the data.
- The longer the lag, the greater the chance that reads will return inconsistent data.

Consistency Model

Tunnable consistency

- The levels of consistency and availability are adjustable to meet certain requirements.
- Individual read and write operations define the number of members or replicas that must acknowledge a request for it to succeed .
- To provide users with a set of tunnable consistency options, MongoDB exposes **read concern** and **write concern** levels.

Consistency Model

Tunnable consistency – read concern

- Determines what level of consistency should be guaranteed during read operations.
  - **Local**: returns data from the instance without garanteeing the data has been written to a majority of the instances.
  - **Majority**: garantees that a majority of the cluster members acknowledges the request.

Consistency Model

Tunnable consistency – write concern

- Establishes the number of replica nodes that must confirm a write operation before it is considered successful.
  - **0**: does not require an acknowledgment of the write.
  - **1**: requires acknowlegment from the primary member only.
  - **N (<number>):** check if the operation has been replicated to the specified number of instances.
  - **Majority**: check if the operations have been propagated to the majority.

Considerations on the dataset

Meteorological data

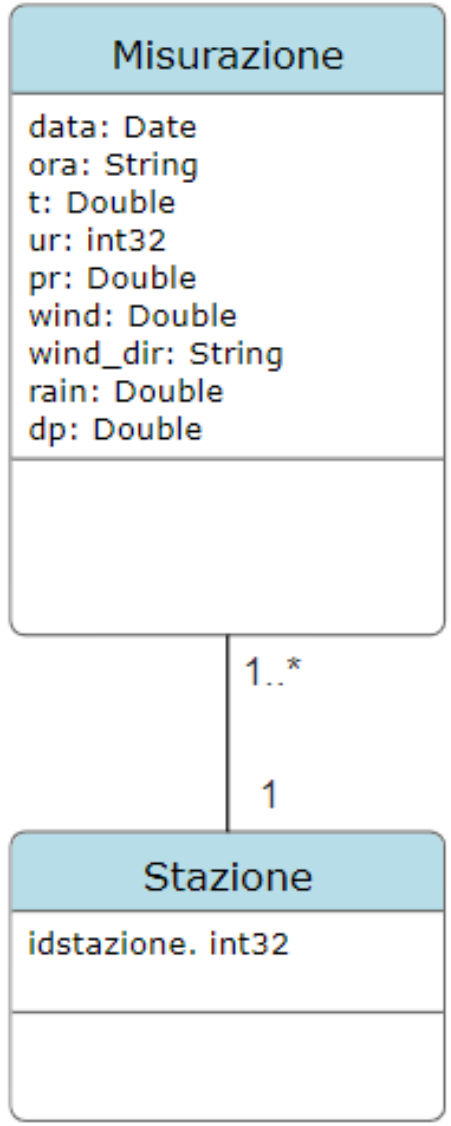
- The choosen dataset is ‘’Meteorological Data’’ which contains a collection of Verona meteorological data from 01-01-2014 to 31-12-2021.
- Data are stored in documents of 10 fields ‘’data, ora, t => temperature, ur => umidità relativa, pr => pressione, wind => velocità del vento, wind_dir => direzione del vento, rain => quantità di pioggia, dp => ignore, idstazione => identificatore della stazione meteo’’.
- It was not necessary to perform any verification operations on the dataset given the **flexible schema** of MongoDB.

data,ora,t,ur,pr,wind,wind_dir,rain,dp,idstazione
2016-02-08,11.54,10.7,82,1015.5,4.26,ESE,10.30,7.8,243
2016-02-08,11.50,9.0,97,1015.5,7.97,SSW,19.80,8.5,351
2016-02-08,11.53,8.7,91,1020.4,5.00,S,10.20,7.3,280
2016-02-08,11.48,8.5,67,1015.0,9.45,SSE,8.10,2.7,343
2016-02-08,11.51,9.6,41,1016.3,0.74,S,4.20,-3.0,368
2016-02-08,11.55,12.0,75,1021.9,15.57,W,0.40,7.7,297
2016-02-08,11.57,8.1,98,1014.1,4.80,SW,0.00,7.8,44
2016-02-08,11.54,8.1,83,1016.6,4.00,NNE,0.00,5.4,292
2016-02-08,11.53,6.3,94,1016.7,0.00,SW,7.20,5.4,324
2016-02-08,11.53,13.8,62,1019.5,3.10,SW,0.00,6.6,267
2016-02-08,11.54,14.9,60,1019.7,3.89,WSW,0.00,7.2,418
2016-02-08,11.53,5.9,91,1017.4,5.93,W,7.00,4.6,102
2016-02-08,11.53,10.4,84,1016.4,7.90,N,0.00,7.8,121
2016-02-08,11.53,9.7,89,1016.1,11.30,NNW,0.00,7.9,106
2016-02-08,11.53,9.6,87,1016.2,9.70,NNW,0.00,7.5,105
2016-02-08,11.53,17.8,60,1020.6,4.45,SSW,0.00,10.0,183
2016-02-08,11.53,17.4,61,1020.3,3.20,SSW,0.00,9.8,422
2016-02-08,11.52,6.8,97,1013.6,3.15,WSW,7.60,6.4,429
2016-02-08,11.53,5.9,97,1015.2,0.00,N,15.40,5.5,140
2016-02-08,11.54,1.3,98,1015.5,3.71,SSE,4.40,1.1,228
2016-02-08,11.53,10.8,81,1017.0,0.00,S,0.00,7.6,378
2016-02-08,11.52,2.1,96,1005.6,0.00,E,0.50,1.5,260
2016-02-08,11.53,4.2,88,999.9,0.00,ESE,0.00,2.4,85

Conceptual Data Model

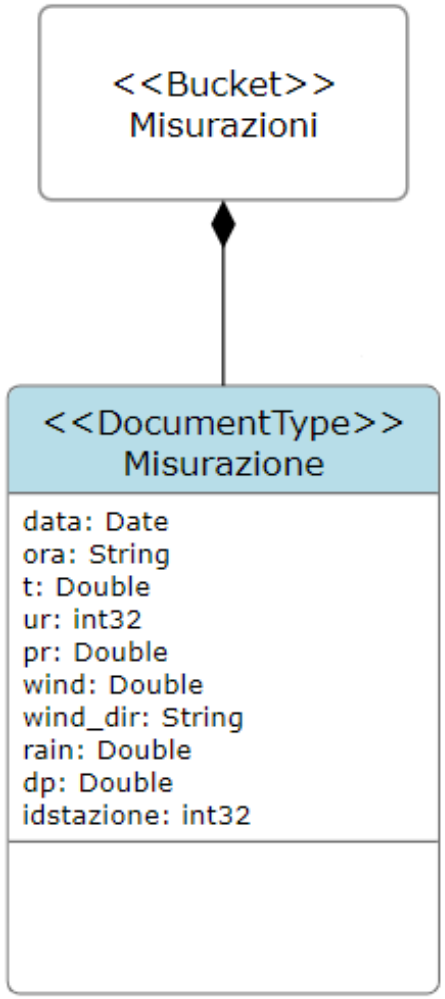
UML Schema

- Since the conceptual schema is independent of the implementation technology, to provide the conceptual data model, we thought about the fact that, data is measured from a station, and a station could collect one or more data, so we have two “collections” with multiple relationships: “Misurazione”, “Stazione”.
- ***data*** *=> Data,* ***ora*** *=> Time,* ***t*** *=> Temperature,* ***ur*** *=> Relative humidity,* ***pr*** *=> Pressure,* ***wind*** *=> Wind speed,* ***wind_dir*** *=> Wind direction,* ***rain*** *=> Amount of rain,* ***idstazione*** *=> Weather station identifier.*



Physical Data Model

- In the physical model, a bucket containing documents related to different measurements has been created.



Queries

Query 1

- Q1: Given a month of a year and the id of a station, find the average, minimum and maximum precipitation value on days with temperature > 30.

```
db.misurazioni.aggregate([
      { $match: {
            data: {$gte: ISODate ("2016-07-01"), $lt: ISODate("2016-08-01")},
            idstazione: 222
            }
      },
      { $group: {
            _id: "$data",
            maxTemperatura:  {$max: "$t"},
            maxRain:  {$max: "$rain"},
            minRain:  {$min: "$rain"},
            avgRain: {$avg: "$rain"}
            }
      },
      {$match: {
            maxTemperatura:  {$gt : 30}
            }
      },
      {$project: {
            _id: 1,
            maxRain:  1,
            minRain:  1,
            avgRain: 1
            }
      },
      {
        $sort: {_id:1}
        }
])
```

Queries

Query 2

- Q2: Find the id of the station that recorded the maximum precipitation value in a given period.

```
db.misurazioni.aggregate  ([
   { $match: {
         data: {
            $gte: ISODate ("2014-01-01"),
            $lt: ISODate("2014-02-01")
            }
         }
   },
   {
     $group: {
         _id: "$idstazione",
         maxRain: {$max: "$rain" }
         }
   },
   {
      $sort: {maxRain: -1}
   },
   {
      $limit: 1
      }
])
```

Queries

Query 3

- Q3: Find the pairs of stations (id1,id2) that recorded the same temperature on some day, reporting not only the pair of ids but also the day on which this occurred, and the rainfall recorded on that day.

```
db.misurazioni.aggregate([
  {
    $group: {
      _id: {data: "$data", temperatura: "$t"},
      stations: {$addToSet: "$idstazione"},
      rain: {$push: "$rain"},
      count: {$sum: 1}
      }
  },
  {
    $match: {
      count: { $gt: 1}
      }
  },
  {
    $project: {
      _id: 0,
      stationPair: "$stations",
      commonDate: "$_id.data",
      rain: 1
    }
  },
  {
    $sort: {
        commonDate: 1
     }
  }
])
```