DBA BRASIL
DATA & CLOUD

Mario Barduchi
mario.barduchi@dell.com
Sr. Database Systems Engineer – LATAM
Oracle ACE Associate

MongoDB for Beginners

Por onde começar a se libertar das limitações do mundo relacional!!!

June, 2023

Who am I?

- I've worked in IT for more than 23 years. I'm Oracle Database Administrator since 2003. MongoDB enthusiast since 2022.
- Sr. Database Systems Engineer at Dell Technologies.
- B.S. in Software Engineering from Metodista University. I completed a graduation course in Oracle Database Administration from FIAP. Currently, I'm pursuing an MBA in Cloud Architecture and Engineering from FIAP.
- I'm Oracle ACE Associate since 2023. I'm the coordinator and organizer of DBA Brasil Data & Cloud and I'm part of GUOB's board.

LinkdIn ***Oracle ACE*** ***DBTips***

Diamante

AGGRANDIZE

COMMVAULT

TD SYNNEX

Platina

GOLDENGATEBR
Consultoria & Treinamentos

DISCOVER

Ouro

scansource

VERTICA
by opentext

Prata

TRACES

CAFÉ COM CLOUD

ROX
We take care of your data

Apoio

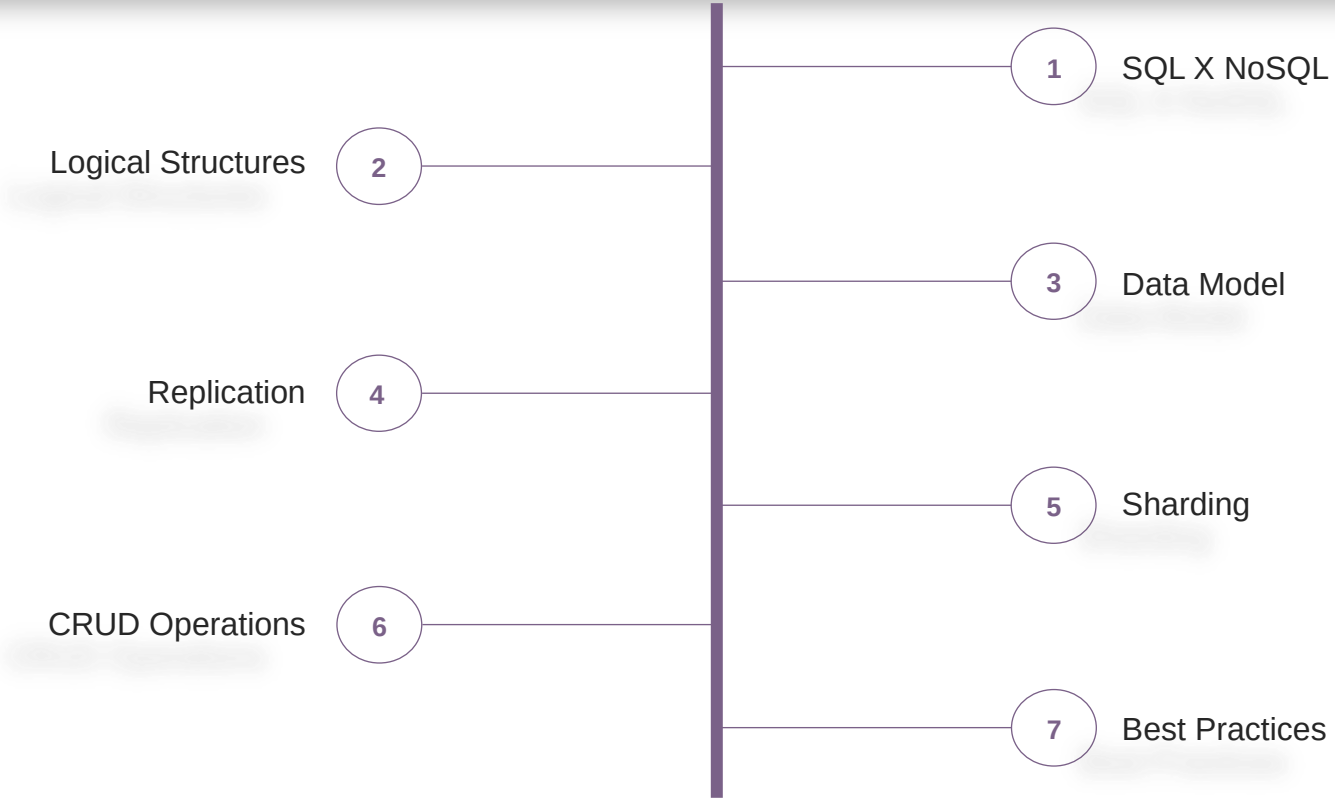
FIAP

GRUPO POSEIDON DIGITAL

DELL Technologies

DBA BRASIL
DATA & CLOUD

Agenda



DELL Technologies

SQL X NoSQL

SQL

SQL is relational. SQL uses tables and indexes structure for data storage.

It's based on a data model that organizes information into rows and columns within tables.

Tables has relationship with each other using Primary Keys and Foreingn Keys.

Transational Databases:

ORACLE

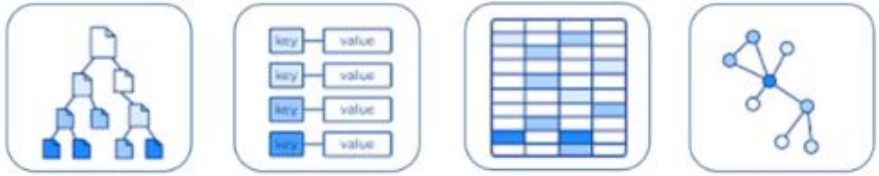
PostgreSQL

MySQL

Microsoft SQL Server

NoSQL

NoSQL doesn't use a table model. NoSQL uses data models based on:



NoSQL is designed for scalability, high availability and flexibility.

MongoDB uses BSON (Binary JSON). It's a binary representation to store data in JSON format, optimized for speed, space, and efficiency.

Fonte: NoSQL Data Models Types: Concepts

SQL X NoSQL

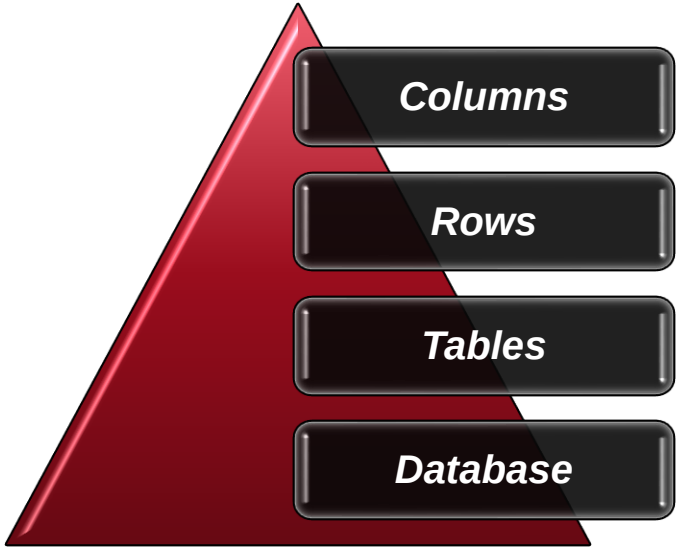
Pros X Cons

Pros of SQL	Cons of SQL
Structured data model for handling data with defined relationships.	Less flexible and requires a predefined schema/data structure.
Mature technology with a wide range of tools and support to manage data.	May not be suitable for specific data types, such as unstructured or semi-structured data
Strong community support and established best practices.	Limited scalability compared to NoSQL.
ACID compliance ensures data consistency and accuracy.	Can be slower than NoSQL for large-scale data processing.
Strong data security and user access controls.	Can be more complex and time-consuming to set up and maintain.

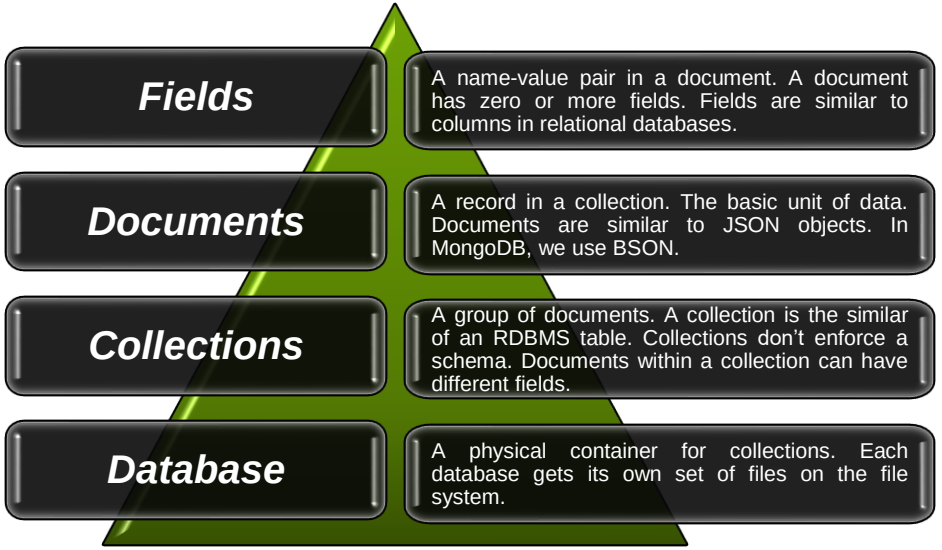
Pros of NoSQL	Cons of NoSQL
Flexible data models for handling unstructured data.	Not as mature as SQL, lacking standardization across different NoSQL databases.
Horizontal scalability and distributed database computing capabilities.	Limited query functionality to manage data compared to SQL.
Easier to maintain compared to SQL.	Less suited for handling complex, interrelated data relationships.
Highly scalable and can handle large volumes of data.	Not ideal for complex transactions that require complex (atomicity, consistency, isolation, durability) compliance.
Ability to handle multiple data formats: JSON, key-value, etc.	Limited reporting and analysis capabilities compared to SQL.

Fonte: NoSQL vs SQL- 4 Reasons Why NoSQL is better for Big Data applications

Logical Structures



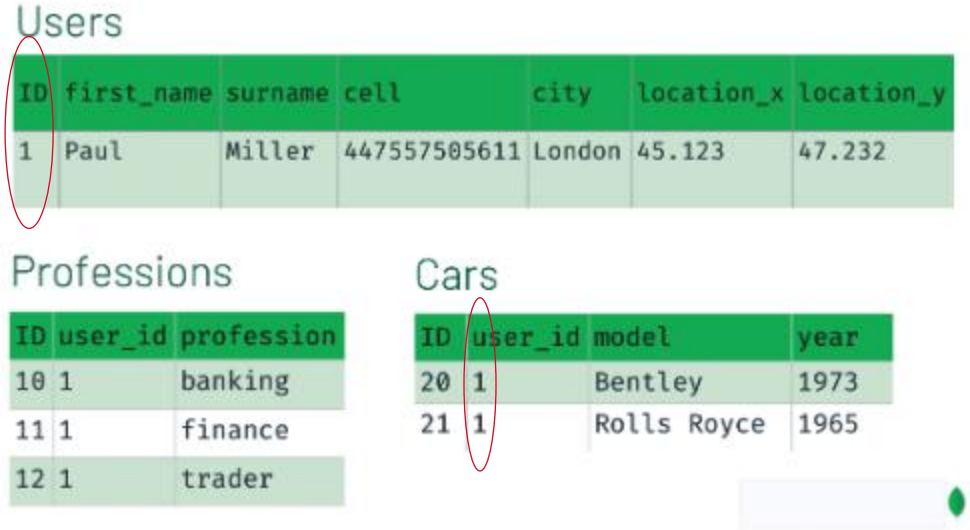
Transacional



MongoDB

Fonte: [Glossary — MongoDB Manual]

Data Model



Users

ID	first_name	surname	cell	city	location_x	location_y
1	Paul	Miller	447557505611	London	45.123	47.232

Professions

ID	user_id	profession
10	1	banking
11	1	finance
12	1	trader

Cars

ID	user_id	model	year
20	1	Bentley	1973
21	1	Rolls Royce	1965

Fonte: MongoDB Schema Design Best Practices | MongoDB

When designing a relational schema:

- ✓ Data model their schema independent of queries.
- ✓ Normalize (typically in 3rd normal form).
- ✓ The normalization will split up your data into tables, so you don't duplicate data.

Example:

```
SELECT
    u.id, u.first_name, u.cell, u.city,
    c.model, c.year
FROM
     users u
JOIN cars c ON c.user_id = u.id
WHERE u.id = 1
ORDER BY 1;
```

Data Model

When you are designing your MongoDB schema, the only thing that matters is that you design a schema that will work well for your application.

When designing a schema, we want to take into consideration the following:

- ✓ No formal process
- ✓ No algorithms
- ✓ No rules
- ✓ Store the data
- ✓ Provide good query performance
- ✓ Require minimum amount of hardware

Fonte: MongoDB Schema Design Best Practices | MongoDB

```
{
   "first_name": "Paul",
   "surname": "Miller",
   "cell": "447557505611",
   "city": "London",
   "location": [45.123, 47.232],
   "profession": ["banking", "finance", "trader"],
   "cars": [
      {
         "model": "Bentley",
         "year": 1973
      },
      {
         "model": "Rolls Royce",
         "year": 1965
      }
   ]
}
```

Data Model

Embedding vs. Referencing

Embedding

You can either embed that data directly.

Pros of Embedding	Cons of Embedding
You can retrieve all relevant information in a single query.	Large documents mean more overhead if most fields are not relevant.
Avoid implementing joins in application code or using $lookup.	You can increase query performance by limiting the size of the documents.
Update related information as a single atomic operation. By default, all CRUD operations on a single document are ACID compliant.	There is a 16MB document size limit in MongoDB. If you are embedding too much data inside a single document, you could potentially hit this limit.
However, if you need a transaction across multiple operations, you can use the transaction operator.	You need to use Upsert (Find + Update) to change the document or to put new information about cars or profession.

Fonte: MongoDB Schema Design Best Practices | MongoDB

```
db.users.insertOne(
{ "_id": 1,
  "name": "Mario Barduchi",
  "addresses": [
    {
                    "type": "Home",
                    "address_1": "288 Guarani Street",
                    "address_2": "Vila Tupi",
                    "city": "SBC",
                    "zip_code": "09760100"
    },
    {
                    "type": "Work",
                    "address_1": "1488 Verbo Divino Street",
                    "address_2": "Santo Amaro",
                    "city": "SP",
                    "zip_code": "05060100"
    }
]})
```

Data Model

Embedding vs. Referencing

Referencing

You can reference another piece of data using the $lookup operator (similar to a JOIN).

Pros of Referencing	Cons of Referencing
By splitting up data, you will have smaller documents.	In order to retrieve all the data in the referenced documents, a minimum of two queries or $lookup required to retrieve all the information.
Less likely to reach 16MB per document limit.	More complexity in the Data Model.
Infrequently accessed information not needed on every query.	More risks for ACID non-compliance.
Reduce the amount of duplication of data. However, it's important to note that data duplication should not be avoided if it results in a better schema.	

Fonte: MongoDB Schema Design Best Practices | MongoDB

```
db.address2.insertOne({
        "_id": 1000, "type": "Home",
        "address_1": "288 Guarani Street",
        "address_2": "Vila Tupi",
        "city": "SBC", "zip_code": "09760100"
})

db.address2.insertOne({
        "_id": 1001, "type": "Work",
        "address_1": "1488 Verbo Divino Street",
        "address_2": "Santo Amaro",
        "city": "SP", "zip_code": "05060100"
})

db.users2.insertOne({
  "_id": 1,
  "name": "Mario Barduchi",
  "addresses":[1000, 1001]
})
```

Modelagem de dados

***One-to-One** (Prefer key value pairs within the doc)*

```
{  "_id": "ObjectId('AAA')",
    "name": "Joe Karlsson",
    "cpf": "999.999.999-01"
}
```

***One-to-Few** (Prefer embedding for one-to-few Relationships)*

```
{  "_id": "ObjectId('XXX')",
   "name": "Mario Barduchi",
   “cpf": "999.999.999-01"
   "addresses": [
        { "type": "Home", "city": "SBC"},
        { "type": "Work", "city": "SP" }]
}
```

Fonte: MongoDB Schema Design Best Practices | MongoDB

***One-to-Many** (Prefer embedding)*

Needing to access an object on its own is a compelling reason not to embed it.
Avoid joins/lookups, but don't be afraid to use if they can provide a better schema design.

Product

```
{ "name": "Lamborghini Sián FKP 37 42115 | Technic",
   "manufacturer": "Lego Inc",
   "catalog_number": "9999",
   "parts": ["ObjectID('XXX')", "ObjectID('ZZZ')", "..."]}
```

Parts

```
{ "_id" : "ObjectID('XXX')",
   "partno" : "aaa-123-bbb",
   "name" : "Block A",
   "qty": "107",
   "cost": "0.37",
   "price":"1.00"}
```

Modelagem de dados

***One-to-Squillions** (Prefer Referencing)*

Tracking data within an unbounded array is hard, since we could potentially hit that 16MB limit.

So, instead of tracking the relationship between the host and the message in the host document, we can change the order. By storing the data in the message, we no longer need to worry about an unbounded array.

Hosts
{ **"_id": ObjectID("XXXX"),**
"name": "mario.barduchi.com",
"ipaddr": "999.99.99.99"}

Log Message
{ "time": ISODate("2013-06-11T09:42:41.382Z"),
"message": "DBA Brasil is on fire!",
"host": ObjectID("XXXX")}

Fonte: MongoDB Schema Design Best Practices | MongoDB

***Many-to-Many** (Prefer Referencing)*

Users
{ **"_id": ObjectID("AAF1")**,
"name": "Kate Monster",
"tasks": [ObjectID("ADF9"), ObjectID("AE02"), ObjectID("…")]
}

Tasks
{ **"_id": ObjectID("ADF9")**,
"description": "Write blog post about MongoDB schema design",
"due_date": ISODate("2014-04-01"),
"owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}

{ **"_id": ObjectID("AE02")**,
"description": "Write blog post about MongoDB schema design",
"due_date": ISODate("2014-04-01"),
"owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}

Replication

Replication increases data availability and reliability.

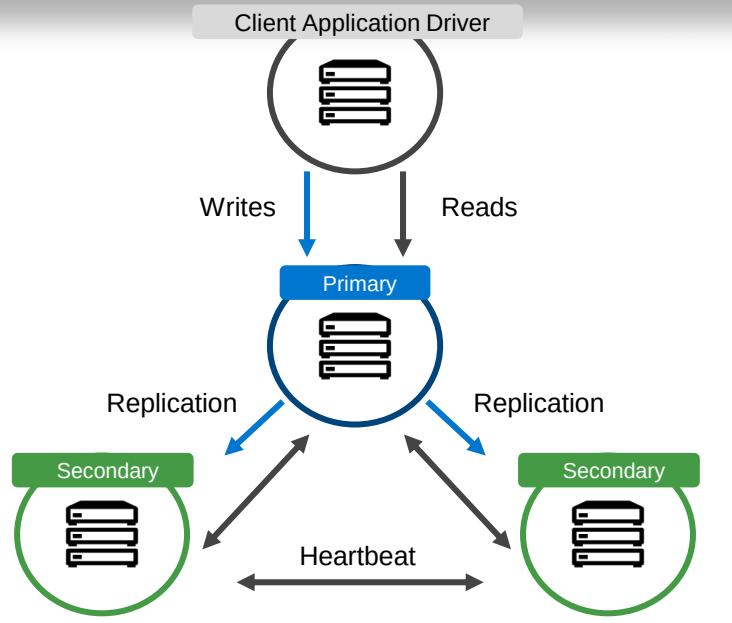
Multiple copies of data maintained in **Replica Sets** using **application level** native replication.

A **Replica Set** is a fully self-healing **Shard.**

A **Replica Set** can have **up to 50 members**, but **only 7 voting members**. If the replica set already has 7 voting members, **additional members must be non-voting members**.

When Primary fails, Secondary nodes “votes” to select primary

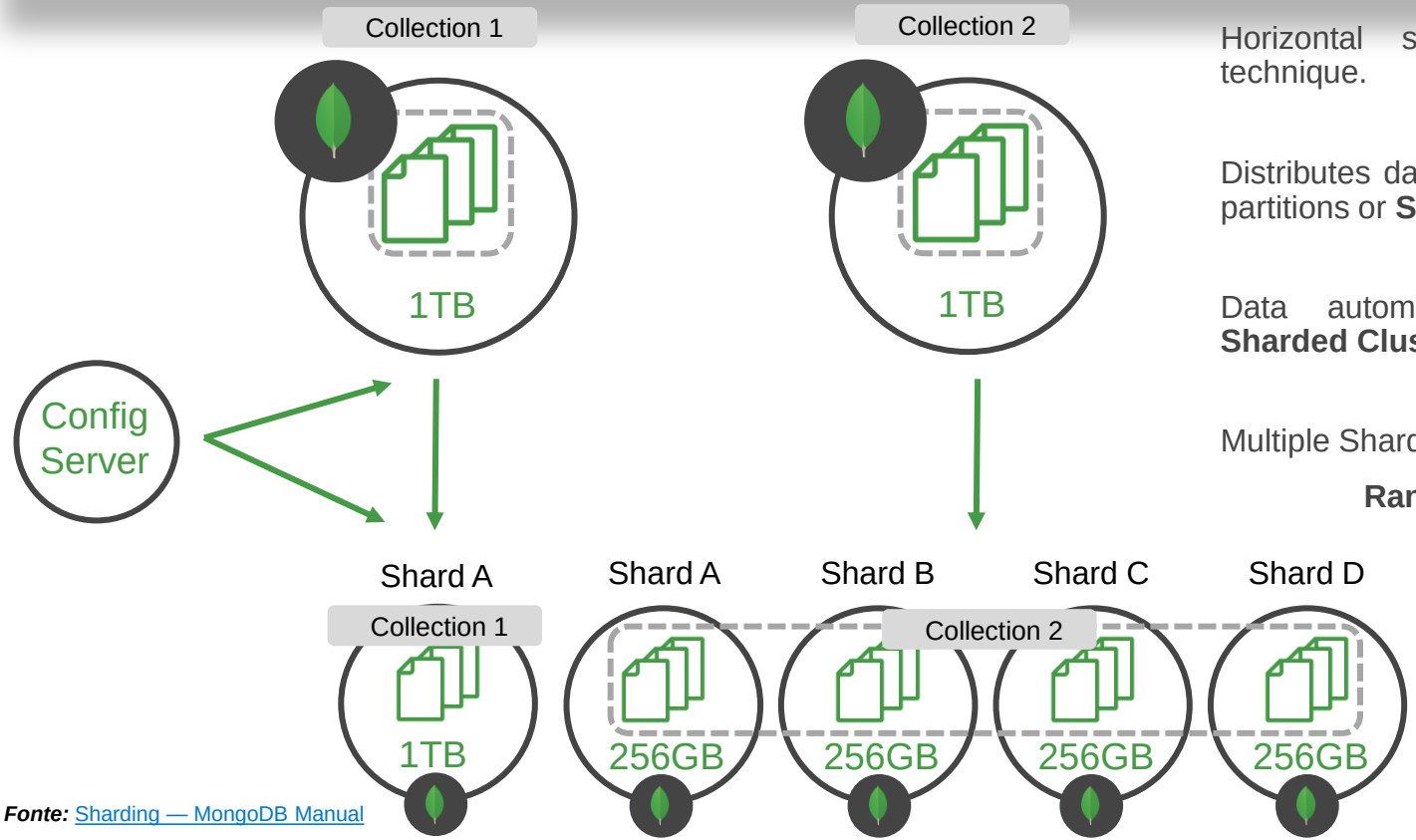
- **10-30 seconds** to declare **primary inaccessible**
- **10-30 seconds** for **election** (cluster unavailable for Writes)



Number of Members	Majority Required to Elect a New Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

Fonte: What Is Replication In MongoDB? | MongoDB and Replica Set Deployment Architectures

Sharding



Horizontal scale-out with **Sharding** technique.

Distributes data across multiple physical partitions or **Shards.**

Data automatically balanced within **Sharded Clusters.**

Multiple Sharding options:

Ranges, Hashes, Zones.

Fonte: Sharding — MongoDB Manual

Sharding + Replication

Clients

Request

Response

MongoDB

Config Servers

- Mongod
- Mongod
- Mongod

Shard1

- Mongod (Primary) — rs0 DB: ZZ Collection: XX
- Mongod (Secondary) — rs0 DB: ZZ Collection: XX
- Mongod (Secondary) — rs0 DB: ZZ Collection: XX

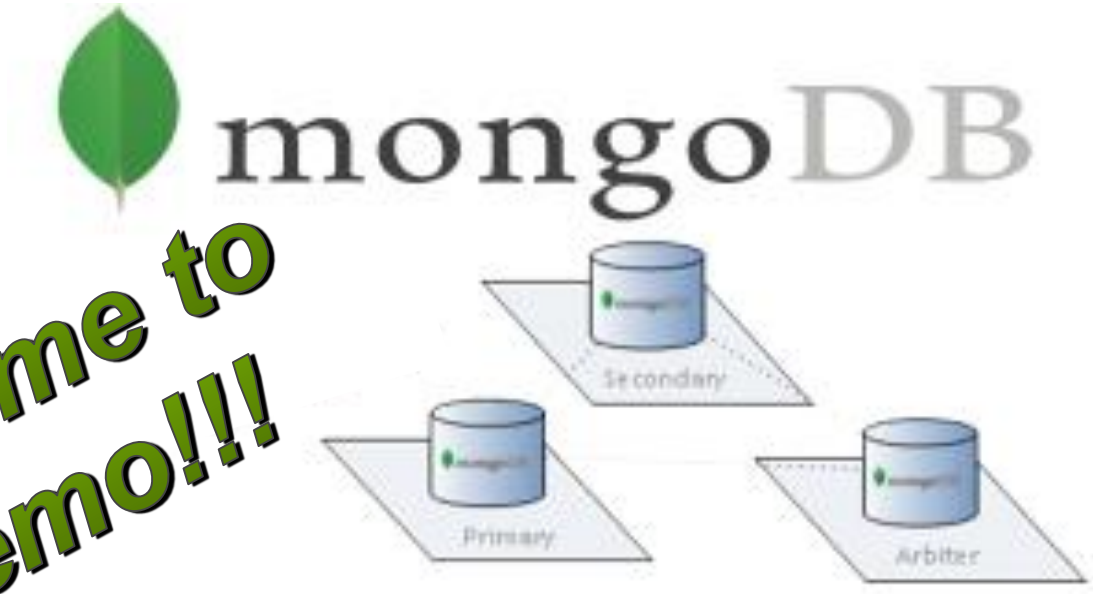
Shard2

- Mongod (Primary) — rs1 DB: ZZ Collection: XX
- Mongod (Secondary) — rs1 DB: ZZ Collection: XX
- Mongod (Secondary) — rs1 DB: ZZ Collection: XX

Shard3

- Mongod (Primary) — rs2 DB: ZZ Collection: XX
- Mongod (Secondary) — rs2 DB: ZZ Collection: XX
- Mongod (Secondary) — rs2 DB: ZZ Collection: XX

CRUD operations



It's time to Demo!!!

CRUD operations

Create

```
use dbabrasil
db.movies.insertOne
(
    {
        codigo: 1, trilogy: "Harry Potter", sequence: "The Philosophers Stone", year: "2001“
    }
)

db.movies.insertMany
(
  [
    {codigo: 2, trilogy: "Harry Potter", sequence: "Harry Potter and the Chamber of Secrets", year: "2002"},
    {codigo: 3, trilogy: "Harry Potter", sequence: "Harry Potter and the Prisoner of Azkaban", year: "2004"},
    {codigo: 4, trilogy: "Harry Potter", sequence: "Harry Potter and the Goblet of Fire", year: "2005"},
    {codigo: 5, trilogy: "Harry Potter", sequence: "Harry Potter and the Order of the Phoenix", year: "2007"}
  ]
)
```

CRUD operations

Create

```
db.movies3.insertOne(
{codigo: 1,
 trilogy: "Harry Potter",
 sequence: [
                    {film: "The Philosophers Stone", year: "2001"},
                    {film: "Harry Potter and the Chamber of Secrets", year: "2002"},
                    {film: "Harry Potter and the Prisoner of Azkaban", year: "2004"},
                    {film: "Harry Potter and the Goblet of Fire", year: "2005"},
                    {film: "Harry Potter and the Order of the Phoenix", year: "2007"},
                    {film: "Harry Potter and the Half-Blood Prince", year: "2009"},
                    {film: "Harry Potter and the Deathly Hallows P1", year: "2010"},
                    {film: "Harry Potter and the Deathly Hallows P2", year: "2011"}
            ]
})
```

CRUD operations

Read

```
-- List all
db.movies.find().pretty()

-- Find the first document matching the object
db.movies.findOne({trilogy: "Transformers"})

-- Limit the number of documents
db.movies.find().limit(3)
```

```
-- Count the number of documents
db.movies.find().count()

-- Where trilogy="Transformers"
db.movies.find( {trilogy: " Transformers "} )

-- List fields _id, codigo,year
db.movies.find( {trilogy:"Transformers"}, { codigo:1,year:1})
```

```
-- List documents with codigo are greather then 5
db.movies.find({ codigo: { $gt: 5 } },{ _id: 0, codigo: 1, year: 1, name: {$concat: ["$trilogy"," - ","$sequence"]} })

-- List documents with codigo are between 5 and 12, Concat Trilogy with Sequence and Sort - Year Desc, Name Asc
db.movies.find({ codigo: { $gt: 5, $lt: 12 } },{codigo: 1, year: 1, name: {$concat: ["$trilogy"," - ","$sequence"]} }).sort({"year": -1, "name": 1 })

-- List documents where sequence.year = 2009
db.movies3.find({ " sequence.year ": '2009' });
```

CRUD operations

Update

```
db.movies.find( { codigo: { $gte: 97 } } ).pretty()

db.movies.updateMany(
        { codigo: { $gte: 97 } } ,{ $set: { year: 2030 } }
)

db.movies.find( { codigo: { $gte: 98 } } ).pretty()
db.movies.updateMany(
        { codigo: 98 } ,{ $set: { year: 2027 } }
)

db.movies3.update(
   { codigo: 90, "sequence.film": "FILM 2“ },
   { $set: {"sequence.$.year": "2029"} } )

db.movies3.updateMany(
   { "sequence.year": "2009“ },
   { $set: {"sequence.$.year": "2049“ } } )
```

Delete

```
db.movies.deleteOne({ codigo: 98 })

db.movies.deleteMany({  trilogy: 'Procurando Nemo' })
```

Best Practices

Understand Schema Differences Between Relational and Document-based Databases - Data modeling

- ✓ Understand your application's query patterns to produces more efficient queries, increases the throughput of insert and update operations, and more effectively distributes your workload across a sharded cluster.
- ✓ Design your data model. Embed Your Data Instead of Relying on Joins.
- ✓ Select the appropriate indexes.
- ✓ Just because MongoDB has a flexible schema does not mean you can ignore schema design!
- ✓ A major advantage of JSON documents is that you have the flexibility.

Memory Sizing: Ensure your working set fits in RAM

- ✓ MongoDB performs best when the application's working set (indexes and most frequently accessed data) fits in memory.
- ✓ RAM size is the most important factor for instance sizing.
- ✓ If price/performance is more of a priority over performance alone, then using fast SSDs to complement smaller amounts of RAM is a viable design choice.
- ✓ When the application's working set fits in RAM, read activity from disk will be low.

Fonte: Performance Best Practices: MongoDB Data Modeling and Memory Sizing | MongoDB

Best Practices

Query patterns and profiling

- ✓ Avoid creating large, unbounded documents.
- ✓ Issue updates to only modify fields that have changed.
- ✓ Update multiple array elements in a single operation.
- ✓ Profile queries with the explain plan:
  - ✓ Which indexes were used.
  - ✓ Whether the query was covered by the index or not.
  - ✓ Whether an in-memory sort was performed, which indicates an index would be beneficial.
  - ✓ The number of index entries scanned.
  - ✓ The number of documents returned, and the number read.
  - ✓ How long the query took to resolve in milliseconds.
  - ✓ Which alternative query plans were rejected (when using the allPlansExecution mode).

Fonte: Performance Best Practices: Query Patterns and Profiling | MongoDB

Best Practices

Indexing

- ✓ Use Compound Indexes (Compound indexes are indexes composed of several different fields).
- ✓ Follow the ESR rule for compound indexes:
  - ✓ Add those fields against which **"Equality"** queries are run.
  - ✓ The next fields to be indexed should reflect the **"Sort"** order of the query.
  - ✓ The last fields represent the **"Range"** of data to be accessed.
- ✓ Use Covered Queries When Possible
  Covered queries return results from an index directly without having to access the source documents and are therefore very efficient. For a query to be covered all the fields needed for filtering, sorting and/or being returned to the client must be present in an index.
- ✓ Use Caution When Considering Indexes on Low-Cardinality Fields.
- ✓ Eliminate Unnecessary Indexes.
- ✓ Use Partial Indexes.
- ✓ Take Advantage of Multi-Key Indexes for Querying Arrays
  If query patterns require accessing individual array elements, use a multi-key index.

Fonte: Performance Best Practices: Indexing | MongoDB

Best Practices

Sharding

- ✓ Sharding for Horizontal Scale Out
- ✓ Sharding Strategies
  - ✓ Ranged Sharding
  - ✓ Hashed Sharding
  - ✓ Zoned Sharding
- ✓ Use Hashed-Based Sharding When Appropriate

Transactions and read/write concerns

- ✓ The Arrival of Multi-Document ACID Transactions
  - ✓ Starting with MongoDB 4.0, support was added for multi-document ACID transactions
- ✓ Best Practices for Multi-Document Transactions
  - ✓ Transaction runtime limit (Default 60 seconds)
  - ✓ Number of operations in a transaction (As a best practice, no more than 1,000 documents)
- ✓ Distributed, multi-shard transactions
- ✓ Choose the Appropriate Write Guarantees
  Write Acknowledged (Default), Journal Acknowledged, Replica Acknowledged and Majority

Fonte: Performance Best Practices: Sharding | MongoDB and Performance Best Practices: Transactions and Read / Write Concerns | MongoDB

Best Practices

Hardware and OS configuration

- ✓ Run on Supported Platforms.
- ✓ Use Multiple CPU Cores
- ✓ Dedicate Each Server to a Single Role in the System
- ✓ Configuring the WiredTiger Cache (WiredTiger storage engine's internal cache)
- ✓ Use Multiple Query Routers
- ✓ Use Interleave Policy on Non-Uniform Memory Access (NUMA) Architecture
- ✓ Storage and Disk I/O
  - ✓ Use NVME for IO Intensive Applications
  - ✓ Use MongoDB's Default Compression for Storage and I/O-Intensive Workloads
  - ✓ Configuring readahead
  - ✓ Use XFS File Systems; Avoid EXT4
  - ✓ Disable Access Time Settings
  - ✓ Disable Transparent Hugepages

Fonte: Performance Best Practices: Hardware and OS Configuration | MongoDB

Best Practices

Benchmarking

- ✓ Use Multiple Parallel Threads
- ✓ Use Bulk Writes

  Similarly, to reduce the overhead from network round trips, you can use bulk writes to load (or update) many documents in one batch.

- ✓ Consider the Ordering of Your Shard Key.
  - ✓ If you configured range based sharding, and load data sorted by the shard key, then all inserts at a given time will necessarily have to go to the same chunk and same shard.
  - ✓ A chunk consists of a range of sharded data.
- ✓ Disable the Balancer for Bulk Loading.
- ✓ Prime the System for Several Minutes.
- ✓ Use Connection Pools.
- ✓ Monitor Everything.

Fonte: Performance Best Practices: Benchmarking | MongoDB

References & Study

MongoDB Docs
[MongoDB Documentation]

MongoDB University
[MongoDB Courses and Trainings | MongoDB University]

MongoDB Atlas
[MongoDB Atlas | MongoDB]

YADAX Blog
[Blog - Yadax]

Q&A

LinkdIn

Oracle ACE

DBTips

Mario Barduchi
mario.barduchi@dell.com
Sr Database Systems Engineer – LATAM

Oracle ACE Associate

DBA BRASIL
DATA & CLOUD

Obrigado