NoSQL

D'Ambrosi Denis

November 15, 2023

Exercise 1

Exercise a.

```
db.video_movieDetails.find({
    "director" : "George Lucas"
})
```

Exercise b.

```
db.video_movieDetails.find(
    { "director" : "George Lucas" },
    { "title" : 1 }
)
```

Exercise c.

```
db.video_movieDetails.find({
    $or : [
        { "director" : null },
        { "director" : "Jon Brewer" }
    ]
})
```

Exercise d.

```
db.video_movieDetails.find({
    $and : [
        { "director" : "Curt McDowell" },
        { "year" : 1980 }
    ]
})
```

Exercise e.

```
db.video_movieDetails.find({
    director: { $regex: /,/ }
})
```

Exercise f.

```
db.video_movieDetails.find({
    "actors" : {
        $all : ["Martin Lawrence", "Will Smith"]
    }
})
```

Exercise g.

```
db.video_movieDetails.find({
    $and : [
        { "actors" :
            { $in : ["Martin Lawrence", "Will Smith"] }
        },
        { "actors" :
            { $not : { $all : ["Martin Lawrence", "Will
                Smith"] } }
        }
    ]
})
```

Exercise 2

Exercise a.

```
db.video_movieDetails.aggregate([
    {
      $project: {
        director: {
          $cond: {
            if: { $regexMatch: { input: "$director",
                regex: /,/ } },
            then: { $split: ["$director", ", "] },
            else: ["$director"]
          }
        }
      }
    },
    {
      $unwind: "$director"
    },
    {
      $group: {
        _id: "$director",
        movieCount: { $sum: 1 }
      }
    },
    {
      $project: {
        director: "$_id",
        movieCount: 1,
        _id: 0
      }
    }
])
```

Exercise b.

```
db.video_movieDetails.updateMany(
    { director: "Roland Emmerich" },
    { $set: { director: "R. Emmerich" } }
)
```

Exercise c.

```
db.video_movies.aggregate([
    {
      $lookup: {
        from: "video_movieDetails",
        localField: "title",
        foreignField: "title",
        as: "movieDetails"
      }
    },
    {
      $unwind: "$movieDetails"
    },
    {
      $match: {
        "movieDetails.countries": "France"
      }
    },
    {
      $project: {
        _id: 0,
        title: 1
      }
    }
])
```

Exercise 3

When run in as a replica set, MongoDB supports various transactional guarantees through its consistency models, in particular it provides tunable consistency options by exposing writeConcern and readConcern levels that can be set on each database operation. For write operations, it offers different writeConcern levels that specify the durability guarantee a write must satisfy before being acknowledged to a client: an operation flagged as $w : N$ will need an acknowledgement by at least N nodes of the data being saved, whereas declaring an operation as $w : majority$ requires that it must be committed by more than half of the servers. Higher write concern levels clearly provide stronger guarantees of permanent durability. MongoDB also supports multi-document transactions with snapshot isolation, where the durability guarantee of any data read or written is deferred until transaction commit time. For read operations, MongoDB provides different readConcern levels, which specify the consistency and durability of the data. The *linearizable* readConcern, when combined with $w : majority$ write operations, offers the strongest consistency guarantees, as we are sure they wil return the effect of the most recent majority write that completed before the read operation began. MongoDB also offers *available* and *snapshot* read concern levels, as well as causally consistent reads. To support these transactional guarantees in a distributed setting, MongoDB utilizes a leader-based consensus protocol similar to the Raft protocol: it allows users to choose the level of consistency they require based on their application's needs, trading off between performance and correctness.

In terms of write concern, higher levels of write concern provide stronger durability guarantees but come with a higher latency cost per operation. On the other hand, lower write concern levels reduce latency but increase the possibility of write loss. Developers often choose weaker write concern levels to minimize latency, especially when write loss is tolerable (such as in a social media application). Similarly, read concern levels determine the consistency guarantees for returned data. Stronger read concern levels provide stronger guarantees but may increase latency and return stale data. Weaker read concerns prioritize fresh data but may risk returning data that is not yet durable. This is clearly less of a problem in non-distributed settings, where there is only a single node database system: the performance impact of durability and consistency guarantees become less significant since there are no network partitions or partial failure modes to consider. Similarly to the features of various ANSI SQL levels, the choice of different options in distributed settings is inherently a trade-off between performance and durability/consistency guarantees. Developers need to carefully consider these trade-offs based on their application requirements in order to use a database that is consistent enough, yet as performant as possible based on the needs of their software.