Contents

0.0.1 1. use("university")

- **What it does:** This command tells MongoDB to use a specific database called "university." Think of a database as a folder that contains related information, like a collection of student records.

0.0.2 Exercise 1

```
db.grades.find(
    {
        scores: {
            $not: {
                $elemMatch: {
                    type: 'homework',
                    score: { $lt: 60 }
                }
            },
            $elemMatch: {
                $or: [
                    { type: 'exam', score: { $gte: 80 } },
                    { type: 'quiz', score: { $gte: 90 } }
                ]
            }
        }
    },
    {
        _id: 0
    }
)
```

- **`db.grades.find(...):`** This command searches for documents (records) in the "grades" collection (like a table) that match certain criteria.
- **`scores:`** This is a field in the documents that contains an array of scores for different types of assessments (like exams, quizzes, and homework).
- **`$not:`** This operator is used to specify that we do NOT want any documents where there is a homework score less than 60.
- **`$elemMatch:`** This is used to match elements in the array. The first `$elemMatch` checks for homework scores, while the second checks for either exam scores of 80 or more or quiz scores of 90 or more.
- **`_id: 0:`** This part specifies that we do not want to include the unique identifier (_id) of the documents in the results.

0.0.3 Exercise 2

```
db.grades.aggregate([
    {
        $match: {
            class_id: { $in: [20, 220, 420] }
        }
    },
    {
        $project: {
            _id: 0,
            student_id: 1,
            class_id: 1,
            min_score: { $min: "$scores.score" },
            max_score: { $max: "$scores.score" },
            avg_score: { $avg: "$scores.score" },
        }
    },
    { $sort: { student: 1, class: 1 } }
])
```

- **`db.grades.aggregate([...]):`** This command processes data in a more complex way than `find`, allowing for calculations and transformations.
- **`$match:`** This stage filters the documents to only include those with a `class_id` of 20, 220, or 420.
- **`$project:`** This stage specifies which fields to include in the output. It calculates the minimum, maximum, and average scores from the `scores` array.
- **`$sort:`** This stage sorts the results by student and class in ascending order.

0.0.4 Exercise 3

```
db.grades.aggregate([
    {
        $project: {
            class_id: 1,
            exam_scores: {
                $filter: {
                    input: "$scores",
                    as: "score",
                    cond: { $eq: ["$$score.type", "exam"] }
                }
            },
            quiz_scores: {
                $filter: {
                    input: "$scores",
                    as: "score",
                    cond: { $eq: ["$$score.type", "quiz"] }
                }
            }
        }
    },
    { $unwind: "$exam_scores" },
    { $unwind: "$quiz_scores" },
    {
        $group: {
            _id: "$class_id",
            max_exam_score: {
                $max: "$exam_scores.score"
            },
            max_quiz_score: {
                $max: "$quiz_scores.score"
            }
        }
    },
    { $sort: { _id: 1 } }
])
```

- **`$project:`** This stage creates new fields for exam and quiz scores by filtering the `scores` array.
- **`$unwind:`** This operator breaks down the arrays into individual documents, allowing for easier calculations.
- **`$group:`** This stage groups the results by `class_id` and calculates the maximum exam and quiz scores for each class.

- **`$sort:`** Finally, it sorts the results by class ID in ascending order.

0.0.5 Exercise 4

```
db.createView(
    "top10students",
    "grades",
    [
        {
            $group: {
                _id: "$student_id",
                avg_score_per_class: {
                    $addToSet: {
                        class: "$class_id",
                        avg_score: { $avg: "$scores.score" }
                    }
                }
            }
        },
        {
            $project: {
                _id: 0,
                student: "$_id",
                avg_score: {
                    $avg: "$avg_score_per_class.avg_score"
                }
            }
        },
        { $sort: { avg_score: -1 } },
        { $limit: 10 },
    ]
)
```

- **`db.createView(...):`** This command creates a view called "top10students" based on the "grades" collection. A view is like a saved query that can be reused.
- **`$group:`** This stage groups the data by `student_id` and calculates the average score for each class.
- **`$project:`** This stage formats the output to show the student ID and their average score across classes.
- **`$sort:`** This sorts the students by their average score in descending order (highest first).
- **`$limit: 10:`** This limits the results to the top 10 students.

0.0.6 Exercise 5

```
db.grades.updateMany(
    { class_id: 339 },
    [
        {
            $set: {
                score_avg: { $avg: "$scores.score" },
                letter: {
                    $switch: {
                        branches: [
                            {
                                case: { $lt: ["$score_avg", 60] },
                                then: "NA"
                            },
                            {
                                case: {
                                    $and: [{ $gte: ["$score_avg", 60] }, { $lt: ["$score_
                                },
                                then: "A"
                            },
                            {
                                case: { $gte: ["$score_avg", 80] },
                                then: "P"
                            }
                        ],
                        default: "",
                    }
                }
            }
        }
    ]
)
```

- **`db.grades.updateMany(...):`** This command updates multiple documents in the "grades" collection that match the criteria.
- **`{ class_id: 339 }:`** This specifies that we want to update documents where the `class_id` is 339.
- **`$set:`** This operator is used to add or update fields in the documents.
- **`score_avg:`** This calculates the average score from the **`scores`** array.
- **`letter:`** This uses a **`$switch`** statement to assign a letter grade based on the average score:
  - "NA" for scores below 60,

– "A" for scores between 60 and 80,
– "P" for scores 80 and above.

0.0.7 Exercise 6a

```
db.runCommand({
    collMod: "grades",
    validator: {
        $jsonSchema: {
            required: ["student_id", "scores", "class_id"],
            properties: {
                student_id: { bsonType: "int" },
                scores: {
                    bsonType: "array",
                    items: {
                        bsonType: "object",
                        required: ["type", "score"],
                        properties: {
                            type: { enum: ["exam", "quiz", "homework"] },
                            score: {
                                bsonType: "double",
                                minimum: Double(0),
                                maximum: Double(100)
                            }
                        }
                    }
                },
                class_id: { bsonType: "int" }
            }
        },
    }
})
```

- **`db.runCommand({...})`:** This command modifies the "grades" collection to enforce rules about the data it can contain.
- **`validator`:** This specifies the rules for the data structure.
- **`$jsonSchema`:** This defines the required fields and their types:
  - `student_id` and `class_id` must be integers.
  - `scores` must be an array of objects, each containing a `type` (which must be "exam," "quiz," or "homework") and a `score` (which must be a number between 0 and 100).

0.0.8 Exercise 6b

```
// First failure
db.grades.insertOne({
    student_id: 100,
    class_id: 120,
    scores: [
        { type: "exam", score: "P" },
        { type: "quiz", score: 99.1 }
    ]
})

// Second failure
db.grades.insertOne({
    student_id: '100',
    class_id: 120,
    scores: [
        { type: "exam", score: 10.3 },
        { type: "quiz", score: 99.1 }
    ]
})

// Success
db.grades.insertOne({
    student_id: 100,
    class_id: 120,
    scores: [
        { type: "exam", score: 10.3 },
        { type: "quiz", score: 99.1 }
    ]
})
```

- **First failure:** The first insert tries to add a score with a string ("P") instead of a number for the exam score, which violates the schema.
- **Second failure:** The second insert uses a string for `student_id` instead of an integer, which also violates the schema.
- **Success:** The last insert correctly follows the schema rules, so it succeeds in adding the document to the collection.

0.0.9 Summary

In summary, this MongoDB code performs various operations on a collection of student grades, including searching for specific records, calculating statistics, creating views,

updating records, and enforcing data validation rules. Each command is designed to manipulate and analyze the data in a structured way, ensuring that the information remains consistent and meaningful.