NoSQL Assignment Report

Jerry Huang 202283890036
Kaifen Jin 202283890029
Weichen Guo 202283890026

1. The design concept of our NoSQL

In the given example of a relational database schema, we have a **Student** table, a **Course** table, an **Enrollment** table, and a **Grades** table. To transform this relational database into a non-relational (NoSQL) database, we can take advantage of NoSQL's support for embedding and referencing documents. Unlike SQL, NoSQL does not require strict adherence to a fixed schema, allowing us to embed sub-documents within a document as needed.

In our design, we propose embedding the **Grades** collection as a sub-document within the **Students** document. This structure reflects the real-world scenario where students are primarily interested in the grades they have received for the courses they are enrolled in. By organizing the data this way, students can more easily access their grades directly within their profiles, improving data accessibility and simplifying queries.

We have also added a new field to the **Course** document called **arrangement**. This field is an array of sub-documents, with each sub-document containing the fields **info**, **week**, **time**, and **location**. This design allows instructors to efficiently manage and organize their schedules for delivering the course. By structuring the data in this way, instructors can easily plan their teaching activities and ensure smooth course scheduling.

We have designed another collection named **Enrollment**, which stores all enrollment data for each student. This collection includes three fields: **CourseId**, **StudentId**, and **EnrollmentDate**. The structure is intentionally kept simple to ensure quick query response times. Additionally, the straightforward design makes the collection easy to manage and organize, facilitating efficient data handling and maintenance.

2. The process to eshtablish a Non-relational Batabase

1. Insert the data

   We can use the `insertMany` function to insert all the information about the students, courses, and enrollments into the database in a single operation.

2. Update function

   In the real world, when students first enter the university, their data is initially added to the database without any grades or courses. For this section, we assume they have just completed their first semester and received their grades. We can use the `updateMany` function to update the documents in the `Students` collection accordingly.

3. The query presentation document

All query will be showed in this chapter.

1. **find()**

find() function with no criteria is used to check all document in the collection. All document in the selected collection will appealing in the MongoShell.

search for all document in Students collection:

```
 db.Students.find()
```

Search for all document in Course collection:

```
db.Courses.find()
```

Search for all document in Enrollments collection:

```
db.Enrollments.find()
```

2. **find().count()**

   find().count() function is used to check the number of the documents in the selected collection.

   Check the number of document in Students collection:

   ```
   db.Students.find().count()
   ```

   To check other collection is the same way.

3. **find().limit()**

   find().limit() function is used to limit the number of the output.

   For example, find() function with no criteria will return all document in the selected collection, the number is large, so we set limit function as:

   ```
   db.Students.find().limit(5)
   ```

   The MongoDB will just return the first five document in the Students collection.

4. **find() with criteria**

   We can add a filter in the find() function to limit the document we want to find.

   For example, we want to find all indormation about the student who FirstName is "Frankie", we can set the find() funcction as:

```
db.Students.find({FirstName:"Frankie"})
```

The output in the MongoShell will just return the document of the student who firstName is "Frankie".

The return value:

```
 _id: ObjectId('676817fdcf2448bbf451c5d3'),
 FirstName: 'Frankie',
 LastName: 'Jin',
 Email: 'FJ001@setu.ie',
 PhoneNumber: '100001',
 .
 .
 .
```

We also can set other filter to the find() function:

```
db.Students.find({LastName:"Guo"})
```

This time, we use the last name as the filter, to find the student who last name is "Guo".

The return value is:

```
 _id: ObjectId('676817fdcf2448bbf451c5d5'),
 FirtName: 'Wisdom',
 LastName: 'Guo',
 Email: 'WG003@setu.ie',
 PhoneNumber: '300003',
 .
 .
 .
```

5. **find() with multiple criteria**

We also can add mutiple filter to the find() function:

```
db.Students.find({FirstName:"Frankie"},{{Email:"FJ001@setu.ie"})
```

By this code, we can also find the document for the student who first name is "Frankie" and his email is "FJ001@setu.ie".

One point we should metioned is the multiple criteria should from one document, or the MongoDB will not find the document with two criteria from different documents.

6. **find(), limit the number of fields back**

   From the previous query, we can get all information about the document we want to search for, but how to do, if we just want to check some of the information in the document.

   To achieve this, we can include a projection document with our queries to specify which fields we would like to see in the results.

```
db.Students.find({FirstName:"Frankie"},{FirstName:1,LastName:1})
```

   By using this code, the return value will just contain the first name and last name of the student.

   The return value is:

```
 _id: ObjectId('676817fdcf2448bbf451c5d3'),
 FirstName: 'Frankie',
 LastName: 'Jin'
```

   The _id is always exist, if we do not want to see it, we can set _id:0 to exclude the field.

```
db.Students.find({FirstName:"Frankie"},{FirstName:1,LastName:1,_id:0})
```

   The return value is:

```
FirstName: 'Frankie',
LastName: 'Jin'
```

7. **find(), with $operation**

   1. $eq

      The $eq operation will matches values that are equal to a specified value.

      Assume we want to check all courses which the credit is 3 and include the fields of Course name and the instrcutor.

```
db.Courses.find({Credits:{$eq:3}},{_id:0, CourseName:1, Instructor:1})
```

      The return value is:

```
{
CourseName: 'Linear Algebra',
Instructor: 'Filius Flitwick'
},
{
CourseName: 'Relational Databases',
Instructor: 'Rubeus Hagrid'
},
{
CourseName: 'NoSQL Databases',
Instructor: 'John Organ'
}
```

2. $gt

   The $gt operation matches values that are greater than a specified value.

   Assume we want to check all courses which the creit is greter than 3:

   ```
   db.Courses.find({Credits:{$gt:3}},{_id:0, CourseName:1, Instructor:1})
   ```

   The return value is:

   ```
   {
   CourseName: 'Advanced Mathematics',
   Instructor: 'Minerva McGonagall'
   },
   {
   CourseName: 'Computer Networks',
   Instructor: 'Aurora Sinistra'
   }
   ```

3. $gte

   The $gte operation matches values that are greater than or equal to a specified value.

   Assume we want to check all courses which credit is greater than or equal to 3:

   ```
   db.Courses.find({Credits:{$gte:3}},{_id:0, CourseName:1, Instructor:1})
   ```

   The return value is:

   ```
   {
   CourseName: 'Advanced Mathematics',
   Instructor: 'Minerva McGonagall'
   ```

```
},
{
CourseName: 'Linear Algebra',
Instructor: 'Filius Flitwick'
},
{
CourseName: 'Computer Networks',
Instructor: 'Aurora Sinistra'
},
{
CourseName: 'Relational Databases',
Instructor: 'Rubeus Hagrid'
},
{
CourseName: 'NoSQL Databases',
Instructor: 'John Organ'
}
```

We can find that the courses which credit is 3 are include in the return data.

4. $in

The `$in` operation matches any of the values specified in an array.

```
db.Courses.find({"Credits":{$in:[3,4]}},
{_id:0,CourseName:1,Instructor:1, Credits:1})
```

The return value is:

```
{
CourseName: 'Linear Algebra',
Credits: 3,
Instructor: 'Filius Flitwick'
},
{
CourseName: 'Computer Networks',
Credits: 4,
Instructor: 'Aurora Sinistra'
},
{
CourseName: 'Relational Databases',
Credits: 3,
Instructor: 'Rubeus Hagrid'
},
{
CourseName: 'NoSQL Databases',
Credits: 3,
Instructor: 'John Organ'
}
```

We can find all courses which the credit is 3 and 4 are returned.

5. $or

We can use $on to search for different fields in different documents.

```
db.Courses.find({$or:[{Instructor:"Minerva McGonagall"},{Credits:3}]},
{_id:0,Instructor:1,"CourseName":1,"Credits":1})
```

The return value is:

```
{
CourseName: 'Advanced Mathematics',
Credits: 6,
Instructor: 'Minerva McGonagall'
},
{
CourseName: 'Linear Algebra',
Credits: 3,
Instructor: 'Filius Flitwick'
},
{
CourseName: 'Relational Databases',
Credits: 3,
Instructor: 'Rubeus Hagrid'
},
{
CourseName: 'NoSQL Databases',
Credits: 3,
Instructor: 'John Organ'
}
```

We can find the Advanced Mathematics is also fined by $or operation.

6. $lt

This operation matches values that are less than a specified value.

```
db.Courses.find({Credits:{$lt:3}},{_id:0, CourseName:1,
Instructor:1,Credits:1})
```

The return value:

```
{
CourseName: 'Operating System',
Credits: 2,
```

```
Instructor: 'Severus Snape'
}
```

7. $lte

   This operation matches values that are less than or equal to a specified value.

```
db.Courses.find({Credits:{$lte:3}},{_id:0, CourseName:1,
Instructor:1,Credits:1})
```

The return value:

```
{
CourseName: 'Linear Algebra',
Credits: 3,
Instructor: 'Filius Flitwick'
},
{
CourseName: 'Operating System',
Credits: 2,
Instructor: 'Severus Snape'
},
{
CourseName: 'Relational Databases',
Credits: 3,
Instructor: 'Rubeus Hagrid'
},
{
CourseName: 'NoSQL Databases',
Credits: 3,
Instructor: 'John Organ'
}
```

8. $ne

   This operation matches all values that are not equal to a specified value.

```
db.Courses.find({Credits:{$ne:3}},{_id:0, CourseName:1,
Instructor:1,Credits:1})
```

The return value

```
{
CourseName: 'Advanced Mathematics',
Credits: 6,
```

```
Instructor: 'Minerva McGonagall'
},
{
CourseName: 'Operating System',
Credits: 2,
Instructor: 'Severus Snape'
},
{
CourseName: 'Computer Networks',
Credits: 4,
Instructor: 'Aurora Sinistra'
}
```

9. $nin
   This operation matches all values that are not equal to a specified value.

```
db.Courses.find({Credits:{$nin:[3,6]}},{_id:0, CourseName:1,
Instructor:1,Credits:1})
```

The return value

```
{
CourseName: 'Operating System',
Credits: 2,
Instructor: 'Severus Snape'
},
{
CourseName: 'Computer Networks',
Credits: 4,
Instructor: 'Aurora Sinistra'
}
```

10. combine $ operation

    The `$` operators can be combined in various ways to construct complex filters.

```
db.Courses.find({"Credits":{$gt:1,$lt:5}},
{_id:0,"CourseName":1,"Credits":1})
```

```
With this code, we can find courses whose credits are greater than 1
and less than 5. In other words, the courses with credits equal to 2,3
and 4.
```

The return value

```
{
CourseName: 'Linear Algebra',
Credits: 3
},
{
CourseName: 'Operating System',
Credits: 2
},
{
CourseName: 'Computer Networks',
Credits: 4
},
{
CourseName: 'Relational Databases',
Credits: 3
},
{
CourseName: 'NoSQL Databases',
Credits: 3
}
```

8. find().sort

   The sort() function organizes query results based on one or more fields. The syntes is:

   ```
   db.collection.find().sort({ field1: 1, field2: -1 });
   ```

   - 1 indicates ascending order.
   - -1 indicates descending order.

   We can set the function like:

   ```
   db.Students.find( ).sort({FirstName:-1})
   ```

   The output will return all students document in descending order of the first name.

   ```

   ```

9. find() function in an array

   In the document of Students collection, we have an array for sub-documents Socres,which contain the **type** and **score**, we can also operation the sub-documents in an array.

   We can get the value in the sub-document by using the syntex like:

   ```

   ```

```
db.Students.find(
 {
      "Grades.Scores":{
           $elemMatch:{Type:"Homework"}
      }
 },
 {
      _id:0,
      FirstName:1,
      LastName:1,
      "Grades.Scores.Score":1,
      "Grades.Scores.Type":1
 }
)
```

But the return value structure is too complex, so we use the find().count() function to check the number of the return values.

```
db.Students.find(
{
     "Grades.Scores":{
          $elemMatch:{Type:"Homework"}
     }
},
{
     _id:0,
     FirstName:1,
     LastName:1,
     "Grades.Scores.Score":1,
     "Grades.Scores.Type":1
}
).count()
```

The return value is 4.

10. aggregation

    Aggregation is a more advanced and powerful version of the `find()` method. It allows us to perform complex operations on the database. The aggregation process operates as a pipeline consisting of four main stages: `match`, `project`, `group`, and `sort`. After each stage is completed, the transformed data is passed to the next stage in the pipeline.

    The `match` stage functions as a filter, identifying and passing only the qualifying data to the next stage. In the `projection` stage, further operations are performed without modifying the original data. Instead, a copy of the data is created for processing. These operations include calculations such as `sum`, `avg`, `min`, `max`, and `count`. Additionally, you can rename fields for output purposes. For example, if a field in the database is stored as "FirstName," it can be renamed to "first name" in the output for better readability.

We will provide some example to presentation.

1. `$unwind`

   In the previous section, we retrieved each student's scores; however, the returned structure was complex and difficult to interpret. To simplify this, we can separate the sub-documents into individual documents for easier readability and management.

   ```
   db.Students.aggregate([{$unwind:"Grades"}])
   ```

   Using the following code, we split the `Grades` sub-document into separate documents while retaining the corresponding student information:

2. The pipeline of aggregate operation

   The aggregate operation can achieve the same functionality as the `find()` function by incorporating the `$match` stage in the pipeline. First, we use `$unwind` to split the document into individual elements, allowing us to examine the values in each document separately. After `match()`, we can project which field is included and excluded.

   ```
   db.Students.aggregate([
   { $unwind: "$Grades" },
   { $unwind: "$Grades.Scores" },
   {
        $match: {
        "Grades.Scores.Score": { $gt: 90 },
        "Grades.Scores.Type": "homework"
        }
   },
   {
        $project: {
        FirstName: 1,
        LastName: 1,
        "Grades.Scores": 1
        }
   }
   ])
   ```

   we can also add multiple filters in the `match()` part in aggregate operation:

   ```
   db.Students.aggregate([
   { $unwind: "$Grades" },
   { $unwind: "$Grades.Scores" },
   {
        $match: {
             "Grades.Scores.Score": { $gt: 50, $lt: 70 },
             "Grades.Scores.Type": "Homework"  // 注意 "Homework" 首字母大
   写
   ```

```
        }
    },
    {
        $project: {
            _id: 0,
            FirstName: 1,
            LastName: 1,
            "Grades.Scores": 1
        }
    }
    ]);
```

After `match()` and `project()`, we can sort the return value, for example ,we can sort by first name:

```
db.Enrollments.aggregate(
    {
        $match:{"CourseId":ObjectId('6767dd51cf2448bbf451c5d2')}
    },
    {
        $project:{
            "FirstName":1,
            "LastName":1,
            "CourseName":1
        }
    },
    {
        $sort:{"FirstName":-1}
    })
```

In addition to `project()`, there are various pipeline operations available. For example, we can use `count()` to calculate the total number of documents and output the result with a specified field name, such as *totalStudents*.

```
db.Enrollments.aggregate(
    {
        $match:{"CourseId":ObjectId('6767dd51cf2448bbf451c5d1')}
    },
    {
        $count:'TotalStudents'
    }
)
```

We can also achieve some simple calaculation in `project()`, such as `$avg`,`$max` and `$min` and so on... We just give some simple example to explain how it work:

```
db.Students.aggregate([
{ $unwind: "$Grades" },
{ $unwind: "$Grades.Scores" },
{
     $match: { "FirstName":"Frankie"}
},
{
     $project: {
     "Grades.Scores.Type":1,
     "Grades.Scores.ClassID":1,
     averageScore: { $avg: "Grades.Scores.Score" }, // Calculate
average score
     highestScore: { $max: "Grades.Scores.Score" }, // Find the maximum
score
     lowestScore: { $min: "Grades.Scores.Score" }
     }
}
])
```

3. `$group`

`$gruop` opeartion is an important operation in aggregate operation, it is used to group documents and apply aggregate calculations to each group. `$gruop` operation is also suppose calculation like `$avg`, `$min` or `$max`, but the most different place between `project` and `group` is the `project` is not calculate by a specified _id, and `gruop` will first search all documents fit the _id, and do the calculation based on the specified documents.

```
db.Students.aggregate(
     [
          { $unwind: "$Grades" },
          { $unwind: "$Grades.Scores" },
          {
          $match: {
               "Grades.Scores.Type": "Exam",
               "Grades.Scores.Score": { $gte: 80 }
          }
     },
     {
          $group: {
               _id: {
                    first_name: "$FirstName",
                    second_name: "$LastName"
               },
               Scores_of_exam: {
                    $push:{
                         courseId: "$Grades.ClassID",
                         type: "$Grades.Scores.Type",
                         score: "$Grades.Scores.Score"
                    }
               }
```

```
            }
        }
        ]
)
```

By this code, we can have a view for all scores for each students who have 80 score and 80 more scores. And all score is grouped by the first name of each student.

```
{
_id: {
first_name: 'Dylan',
second_name: 'Wu'
},
Scores_of_exam: [
{
     courseId: ObjectId('6767dd51cf2448bbf451c5d2'),
     type: 'Exam',
     score: 80
},
{
          courseId: ObjectId('6767dd51cf2448bbf451c5cd'),
          type: 'Exam',
          score: 98
     }
     ]
},
{
_id: {
first_name: 'Frankie',
second_name: 'Jin'
},
Scores_of_exam: [
{
     courseId: ObjectId('6767dd51cf2448bbf451c5cd'),
     type: 'Exam',
     score: 80
},
{
     courseId: ObjectId('6767dd51cf2448bbf451c5ce'),
     type: 'Exam',
     score: 87
}
]
}
.
.
.
.
```

4. `$lookup`

The `lookup` operation can join the documents in different collection together without change the original information in the documents. In the **Enrollments** collection, we just set three fields **StudentId**, **CourseId** and **EnrollmentDate**, but we can not get the information by the id of students and course, so we need to use the `lookup` opearion to join two collection together.

```
db.Enrollments.aggregate([
     {
          $match: { "CourseId": ObjectId('6767dd51cf2448bbf451c5d2') }
     },
     // connect to Students collection to get the information
     {
          $lookup: {
               from: "Students",
               localField: "StudentId",
               foreignField: "_id",
               as: "StudentInfo"
          }
     },
     {
          $unwind: "$StudentInfo"
     },
     // connect to Course collection to get the information
     {
          $lookup: {
               from: "Courses",
               localField: "CourseId",
               foreignField: "_id",
               as: "CourseInfo"
          }
     },
     {
          $unwind: "$CourseInfo"
     },
     {
          $project: {
               _id: 0,
               FirstName: "$StudentInfo.FirstName",
               LastName: "$StudentInfo.LastName",
               CourseName: "$CourseInfo.CourseName"
          }
     },
     {
          $sort: { "LastName": 1 }
     }
]);
```

5. One practice use example in realistic world

   In addition to the operations covered in lessons, we have explored more advanced aggregation techniques in MongoDB.

Using aggregation, we can project data without modifying the original document.

For example, in `Grades.Scores`, certain courses, such as **Advanced Mathematics**, **Linear Algebra**, and **Operating Systems**, include an extra `homework` score. To handle these cases, we need to separate these courses and calculate their total scores accordingly.

Suppose the function to calculate the total score is:

$$totalScore = Exam \times 0.5 + quiz \times 0.2 + Assignment \times 0.3$$

If the course have homework,the function is:

$$totalScore = Exam \times 0.5 + quiz \times 0.2 + Assignment \times 0.15 + homework \times 0.15$$

The mongoDB code is:

```
db.Students.aggregate([
// unwind Grades and Scores array
{ $unwind: "$Grades" },
{ $unwind: "$Grades.Scores" },
// Mark the "Homework" and "Assignment"
{
     $addFields: {
          isHomework: { $eq: ["$Grades.Scores.Type", "Homework"] },
          isAssignment: { $eq: ["$Grades.Scores.Type", "Assignment"] }
     }
},
// Grouping to detect the presence of Homework
//calculate the sum of scores for Assignment and Assignment
{
     $group: {
          _id: { firstName: "$FirstName", lastName: "$LastName",
classId: "$Grades.ClassID" },
          totalQuizScore: {
               $sum: {
                    $cond: [{ $eq: ["$Grades.Scores.Type", "Quiz"] },
"$Grades.Scores.Score", 0]
               }
          },
          totalExamScore: {
               $sum: {
                    $cond: [{ $eq: ["$Grades.Scores.Type", "Exam"] },
"$Grades.Scores.Score", 0]
               }
          },
          totalAssignmentScore: {
               $sum: {
                    $cond: ["$isAssignment", "$Grades.Scores.Score", 0]
               }
          },
          totalHomeworkScore: {
               $sum: {
                    $cond: ["$isHomework", "$Grades.Scores.Score", 0]
```

```
                }
            },
            hasHomework: {
                $max: {
                    $cond: ["$isHomework", 1, 0]
                }
            }
        }
    },
    // dymatical calculate the weighted grades
    {
        $addFields: {
            weightedAssignmentAndHomework: {
                $cond: [
                    { $eq: ["$hasHomework", 1] },
                    { $add: [
                    { $multiply: ["$totalAssignmentScore", 0.15] },
                    { $multiply: ["$totalHomeworkScore", 0.15] }
                    ] },
                    { $multiply: ["$totalAssignmentScore", 0.3] }
                ]
            },
            weightedQuiz: { $multiply: ["$totalQuizScore", 0.2] }, //
    Quiz weight 20%
            weightedExam: { $multiply: ["$totalExamScore", 0.5] } // Exam
    weight 50%
        }
    },
    {
        $project: {
            _id: 0,
            firstName: "$_id.firstName",
            lastName: "$_id.lastName",
            courseId: "$_id.classId",
            totalScore: {
                $add: ["$weightedAssignmentAndHomework",
    "$weightedQuiz", "$weightedExam"]
                }
            }
        }
        ]);
```

By this code, we can check the total score for each course the student enrolled.

```
{
firstName: 'Jerry',
lastName: 'Huang',
courseId: ObjectId('6767dd51cf2448bbf451c5ce'),
totalScore: 74.85
},
{
firstName: 'Sherlock',
```

```
lastName: 'Xia',
courseId: ObjectId('6767dd51cf2448bbf451c5d1'),
totalScore: 84.2
},
.
.
.
.
.
```

6. The data calculation and out

   In the previous part, we gerenate the total score for each lesson, but the total score will store in the document.
   Aggregate operation also provide the function `$out` to store the gerenated data into a new collection, in this part, we will store the total score for each courses of the students and the basic information of the students into a new collection `StudentsWithTotalScores` .

```
db.Students.aggregate([
{ $unwind: "$Grades" },
{ $unwind: "$Grades.Scores" },
{
$addFields: {
     isHomework: { $eq: ["$Grades.Scores.Type", "Homework"] },
     isAssignment: { $eq: ["$Grades.Scores.Type", "Assignment"] }
}
},
{
$group: {
     _id: { studentId: "$_id", classId: "$Grades.ClassID" },
     firstName: { $first: "$FirstName" },
     lastName: { $first: "$LastName" },
     email: { $first: "$Email" },
     phoneNumber: { $first: "$PhoneNumber" },
     totalQuizScore: {
     $sum: { $cond: [{ $eq: ["$Grades.Scores.Type", "Quiz"] },
"$Grades.Scores.Score", 0] }
     },
     totalExamScore: {
     $sum: { $cond: [{ $eq: ["$Grades.Scores.Type", "Exam"] },
"$Grades.Scores.Score", 0] }
     },
     totalAssignmentScore: {
     $sum: { $cond: ["$isAssignment", "$Grades.Scores.Score", 0] }
     },
     totalHomeworkScore: {
     $sum: { $cond: ["$isHomework", "$Grades.Scores.Score", 0] }
     },
     hasHomework: {
     $max: { $cond: ["$isHomework", 1, 0] }
     }
```

```
}
},
{
$addFields: {
     weightedAssignmentAndHomework: {
     $cond: [
          { $eq: ["$hasHomework", 1] },
          { $add: [
          { $multiply: ["$totalAssignmentScore", 0.15] },
          { $multiply: ["$totalHomeworkScore", 0.15] }
          ] },
          { $multiply: ["$totalAssignmentScore", 0.3] }
     ]
     },
     weightedQuiz: { $multiply: ["$totalQuizScore", 0.2] },
     weightedExam: { $multiply: ["$totalExamScore", 0.5] }
}
},
{
$addFields: {
     totalScore: {
     $add: ["$weightedAssignmentAndHomework", "$weightedQuiz",
"$weightedExam"]
     }
}
},
{
$project: {
     _id: 0,
     studentId: "$_id.studentId",
     classId: "$_id.classId",
     firstName: 1,
     lastName: 1,
     email: 1,
     phoneNumber: 1,
     totalScore: 1
}
},
// Save to a new collection
{
$out: "StudentsWithTotalScores" // Name of the new collection
}
]);
```

Then, we can find a new collection in the database, contain all students basic inforamtion and the total score for their each enrolled leasons.

```
{
_id: ObjectId('6770f0f79110dbacad9af6b7'),
firstName: 'Zane',
lastName: 'Wang',
```

```
email: 'ZW004@setu.ie',
phoneNumber: '400004',
totalScore: 81.8,
studentId: ObjectId('676817fdcf2448bbf451c5d6'),
classId: ObjectId('6767dd51cf2448bbf451c5d1')
},
{
_id: ObjectId('6770f0f79110dbacad9af6b8'),
firstName: 'Frankie',
lastName: 'Jin',
email: 'FJ001@setu.ie',
phoneNumber: '100001',
totalScore: 83.8,
studentId: ObjectId('676817fdcf2448bbf451c5d3'),
classId: ObjectId('6767dd51cf2448bbf451c5ce')
},
.
.
.
.
.
```