MongoDB Queries II

CS 341 Database Systems

MongoDB Query API

- The way you will interact with your data.
- Used in 2 ways
  - CRUD Operations
  - Aggregation Pipelines

Create a Collection

- **db.createCollection(“nameofCollection")**
  - Creates a new collection
  - Can also include schema validation
- **db.students.insertOne(object)**
  - Collection during the insert process

Insert Documents

insertOne()

```
db.posts.insertOne({ title: "Post Title 1", body: "Body of post.", category: "News", likes: 1, tags: ["news", "events"], date: Date() })
```

insertMany()

```
db.posts.insertMany([ {
title: "Post Title 2",
body: "Body of post.",
category: "Event", likes: 2, tags: ["news", "events"], date: Date() },
{ title: "Post Title 3",
body: "Body of post.",
category: "Technology",
likes: 3, tags: ["news", "events"], date: Date() }
])
```

Data Retrieval

- **db.collection.find()**
  - Retrieves data
  - Fetches all documents if left empty
- **db.collection.findOne()**

db.collection.find()

- Enclose in brackets
- First {} contains the query criteria
- Second {} specifies the projected columns (either 1s or 0s, only exception is _id which acts as the PK for the object.
- Within the query criteria, the comparator operators are to be used with **:**

SQL VS Mongo Query API

SQL	Mongo Query
Select * from Students	db.students.find()
Select student_name from Students	db.students.find({}, {“student_name”:1})
Select student_name from Students Where major = ‘CS’	db.students.find({“major”: “CS”}, {“student_name”:1})

Projection

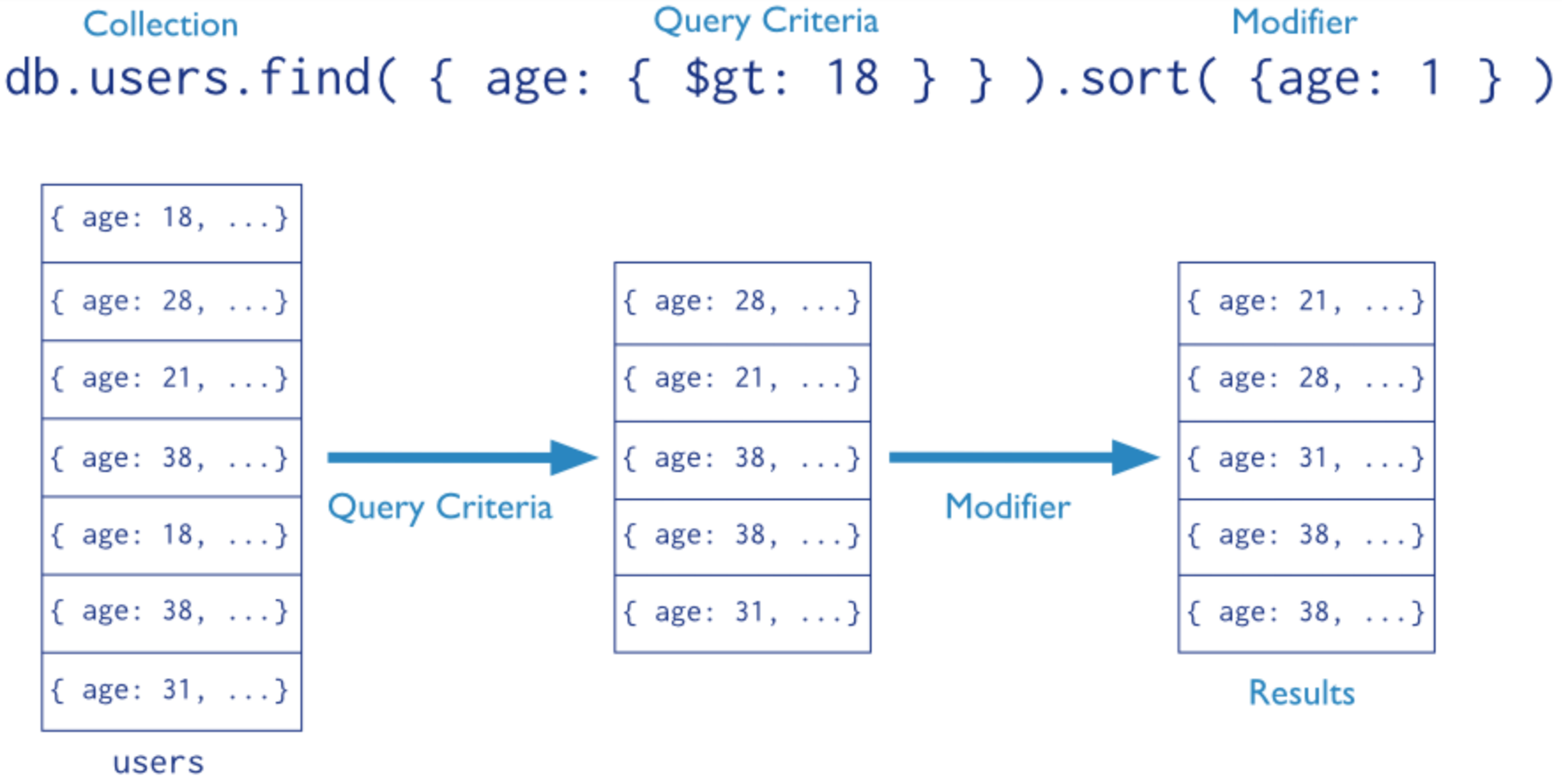
Select student_name from Students Where major = 'CS'	db.students.find({"major": "CS"}, {"student_name":1, "_id":0})

- Note: The _id field is also included. This field is always included unless specifically excluded.
- We use a 1 to include a field and 0 to exclude a field.
- You cannot use both 0 and 1 together except in case of "_id"
- Either specify the fields to include or to exclude.

Comparison Operators

- **$eq** Values are equal
- **$ne** Values are not equal
- **$gt** Value is greater than another value
- **$gte** Value is greater than or equal to another value
- **$lt** Value is less than another value
- **$lte** Value is less than or equal to another value
- **$in** Value is matched within an array

Find the users of age greater than 18 and sort by age.



Logical Operators

- **$and** Returns documents where both queries match
- **$or** Returns documents where either query matches
- **$nor** Returns documents where both queries fail to match
- **$not** Returns documents where the query does not match

Note: In MongoDB, when you provide multiple conditions in the find method, it implicitly interprets them as an AND operation. So, the use of $and is often optional but recommended.

Logical Operators

- db.students.find({gender: “m”, nationality: “Pakistan”})

Is equivalent to:

- db.students.find({$and: [{ gender: "m" },{ nationality: "Pakistan" }]})

- db.musicians.find({$or: [{ instrument: "Drums" }, { born: 1945 }] })

- db.musicians.find({ "instrument": { $in: ["Keyboards", "Bass"] } })

Evaluation Operators

- **$regex** Allows the use of regular expressions when evaluating field values
- **$text** Performs a text search
- **$where** Uses a JavaScript expression to match documents

Regex – contains

- All products that have t in them

```
db.purchase_orders.find( { product: { $regex: "t"} } )
```

- All products that have t in them (case insensitive)

```
db.purchase_orders.find( { product: { $regex: /t/i} } )
```

Regex – starts with

- Product starting with t:

```
db.purchase_orders.find( { product: { $regex: "^t"} } )
```

- Product starting with T (case insensitive)

```
db.purchase_orders.find( { product: { $regex: /^t/i } } )
```

Regex – ends with

Product ending with h:

```
db.purchase_orders.find( { product: { $regex: "h$"} } )
```

Case Insensitive:

```
db.purchase_orders.find( { product: { $regex: /h$/i} } )
```

Regex – start and end

```
db.purchase_orders.find({product:{$regex: /^M.*e$/}})
```

.* → One or Many

. : Matches any single character (except for a newline character).
* : Matches zero or more occurrences of the preceding character or group.

Sort

- Ascending **1**
- Descending -**1**

```
db.students.find({gender: 'm'}).sort({nationality: -1});
```

```
db.students.find({gender: ‘m’}).sort({nationality: -1,
firstName: 1});
```

Limit

- Limit records to just 2

```
db.students.find({gender: ‘f', $or: [{nationality: ‘pakistani'}, {nationality: 'american'}]}).limit(2);
```

- Retrieve the 3rd and 4th record by skipping over first 2.

```
db.students.find({gender: ‘f', $or: [{nationality: ‘pakistani'}, {nationality: 'american'}]}).limit(2).skip(2);
```

Count

Counting number of batteries sold

```
db.purchase_orders.countDocuments({product: “battery"})
```

Count number of documents found

```
db.purchase_orders.find({}).count()
```

Distinct

- List all the distinct products

```
db.purchase_orders.distinct("product")
```

ElemMatch

Element Matching

ElemMatch for Querying Arrays

- The $elemMatch operator matches documents that contain an array field with at **least one element that matches all the specified query criteria.**
- This is primarily used when you want to *match documents based on conditions within an array field.*
- It is useful when you want **all the specified conditions** to be met by **a single element within an array.**
- You don't need to use the $elemMatch operator when specifying a single search condition on an array field. You may directly access the field through dot notation.

ElemMatch in Scores collection

```
{ _id: 1, results: [ 82, 85, 88 ] }
{ _id: 2, results: [ 75, 88, 89 ] }
```

The following query matches only those documents where the results array contains at least one element that is **both** greater than or equal to 80 and is less than 85:

```
db.scores.find(
{ results: { $elemMatch: { $gte: 80, $lt: 85 } } }
)
```

Dot Notation

- Dot notation is used to access fields within nested documents or arrays.
- It helps you navigate through the structure of a document.
- Example *Results.subject.score*

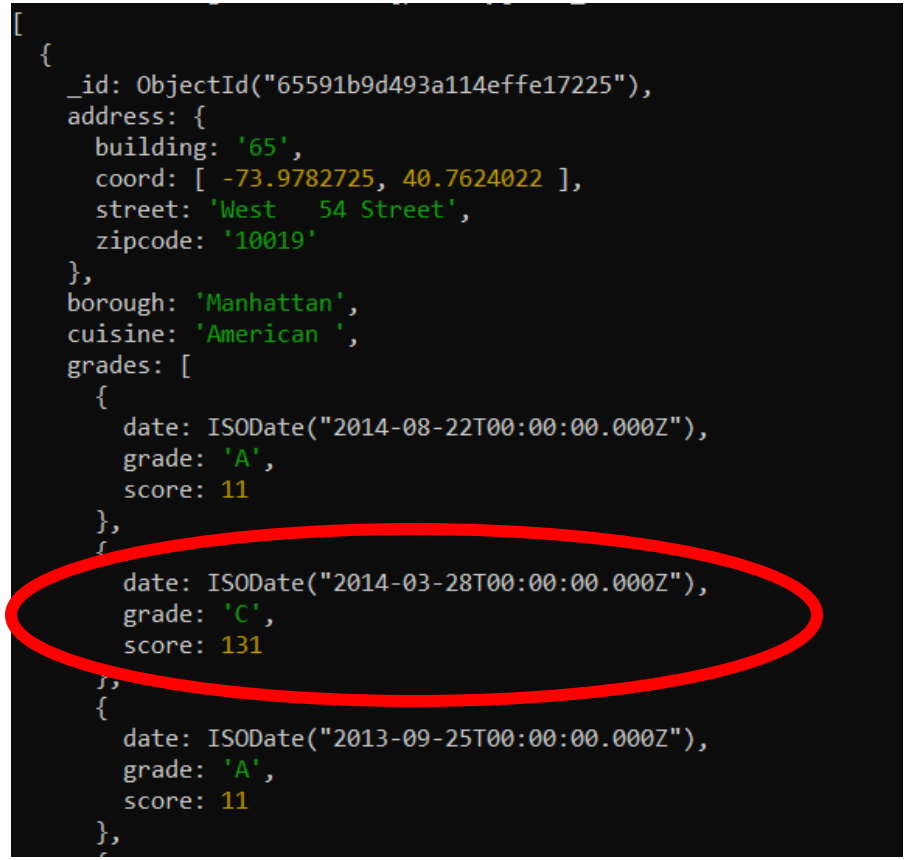
Restaurants Collection

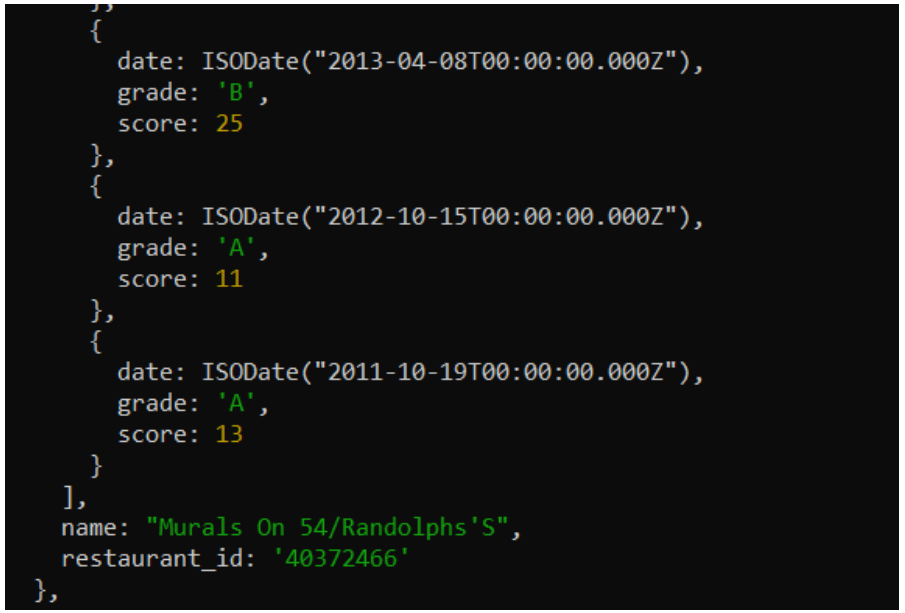

```
{
  "address": {
     "building": "1007",
     "coord": [ -73.856077, 40.848447 ],
     "street": "Morris Park Ave",
     "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",
  "grades": [
     { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
     { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
     { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
     { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
     { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```


Display restaurant names with score more than 80 but less than 100.

date: ISODate("2014-06-27T00:00:00.000Z")

db.restaurants.find({"grades.score": { $gt: 80, $lt :100}});





```
Atlas atlas-9egitm-shard-0 [primary] iba_db> db.restaurants.find({ "grades": { $elemMatch: { "score": { $gt: 80, $lt: 100 } } } }, { "name": 1 }).count()
3
Atlas atlas-9egitm-shard-0 [primary] iba_db> db.restaurants.find({ "grades.score": { $gt: 80, $lt: 100 } }, { "name": 1 }).count()
4
Atlas atlas-9egitm-shard-0 [primary] iba_db>
```

- **db.restaurants.find({grades : { $elemMatch:{"score":{$gt : 80 , $lt :100}}}});**

```
{
  _id: ObjectId("673ceb13d4e1d7875c6d112e"),
  address: {
    building: '',
    coord: [ -74.0163793, 40.7167671 ],
    street: 'Hudson River',
    zipcode: '10282'
  },
  borough: 'Manhattan',
  cuisine: 'American ',
  grades: [
    {
      date: ISODate("2014-06-27T00:00:00.000Z"),
      grade: 'C',
      score: 89
    },
    {
      date: ISODate("2013-06-06T00:00:00.000Z"),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate("2012-06-19T00:00:00.000Z"),
      grade: 'A',
      score: 13
    }
  ],
  name: 'West 79Th Street Boat Basin Cafe',
  restaurant_id: '40756344'
}
```

elemMatch is more Specific

- **Directly on "grades.score":**

```
db.restaurants.find({ "grades.score": { $gt: 80, $lt: 100 } }, { "name": 1 })
```

- This query will return documents where there is at least one grade within the "grades" array with a score in range 80 and less than 100. However, it doesn't ensure that both conditions are satisfied by the same grade element within the array.
- **Using $elemMatch:**

```
db.restaurants.find({ "grades": { $elemMatch: { "score": { $gt: 80, $lt: 100 } } } }, { "name": 1 })
```

- This query will return documents where there is at least one grade within the "grades" array that has a score greater than 80 and less than 100. It ensures that a single grade within the array satisfies both conditions simultaneously.

$all operator

- $all is used to match array fields where the array contains all specified elements. It can also be combined with $elemMatch to apply specific conditions to elements in the array.
- *Q. Find documents where there is at least one grade with score 10 and at least one grade with grade 'A'*
- Notice in our dataset, a score of 10 would imply a grade A hence the condition would appear to be correct in a single array element. Let's consider a smaller and different version of the restaurants collection

```
{
  "name": "Restaurant A",
  "grades": [
    { "score": 10, "grade": "B" },
    { "score": 8, "grade": "A" },
    { "score": 5, "grade": "C" }
  ]
},
{
  "name": "Restaurant B",
  "grades": [
    { "score": 10, "grade": "A" },
    { "score": 6, "grade": "B" }
  ]
},
{
  "name": "Restaurant C",
  "grades": [
    { "score": 7, "grade": "B" },
    { "score": 8, "grade": "C" }
  ]
}
```

```
db.restaurants.find({
  "grades": {
    $all: [
      { $elemMatch: { "score": 10 } },
      { $elemMatch: { "grade": "A" } }
    ]
  }
})
```

- When using $all with $elemMatch, we are specifying that the array must contain elements such that both conditions are satisfied, but the conditions do not need to be met by the same element.

```
[
        { "name": "Restaurant A" },
        { "name": "Restaurant B" }
]
```

Find documents where all elements in the grades array have a score of 10 and a grade of 'A'

```
{
   "grades": {
     "$not": {
       "$elemMatch": {
         "$or": [
           { "score": { "$ne": 10 } },
           { "grade": { "$ne": "A" } }
         ]
       }
     }
   }
}
```

MongoDB Queries II

Delete the users with status equal to D.

SQL

```
DELETE FROM users      <-- table
WHERE  status = 'D'    <-- delete criteria
```

MongoDB

```
db.users.remove(       <-- collection
   { status: "D" }     <-- remove criteria
)
```

Delete

- Delete the first document where a condition meets:
  db.posts.deleteOne({ title: "Post Title 5" })

- Delete all documents: **db.collection.deleteMany({ })**

- Delete all Technology posts: **db.posts.deleteMany({ category: "Technology" })**

- REMOVE: Old but works in the same manner. Can also be modified to operate as deleteOne or delete Many: **db.persons.remove({first: 'james', last: 'robert'});**

Update

updateOne()

- **db.posts.find({ title: "Post Title 1" })**
- **db.posts.updateOne({ title: "Post Title 1" }, { $set: { likes: 2 } })**
- View again to see the updated post.


```
{
  "_id": ObjectId("607f1f77bcf86cd799439011"),
  "title": "Post Title 1",
  "content": "This is the first post.",
  "likes": 0
}
```

```
{
  "_id": ObjectId("607f1f77bcf86cd799439011"),
  "title": "Post Title 1",
  "content": "This is the first post.",
  "likes": 2
}
```

updateMany()

Update likes on all documents and increase them by 1.

For this we will use the **$inc (increment)** operator:

db.posts.updateMany({}, { $inc: { likes: 1 } })

```
[
  { "_id": 1, "title": "Post Title 1", "likes": 3 },
  { "_id": 2, "title": "Post Title 2", "likes": 5 },
  { "_id": 3, "title": "Post Title 3", "likes": 2 }
]
```

Update the users of age greater than 18 by setting the status field to A.

SQL

```
UPDATE users          <-- table
SET    status = 'A'   <-- update action
WHERE  age > 18       <-- update criteria
```

MongoDB

```
db.users.update(                    <-- collection
   { age: { $gt: 18 } },            <-- update criteria
   { $set: { status: "A" } },       <-- update action
   { multi: true }                  <-- update option
)
```

MULTI.
If set to true, updates multiple documents that meet the query criteria. If set to false, updates one document. The default value is false.

Aggregation Pipeline

Aggregation Pipeline

- Aggregation operations allow you to group, sort, perform calculations, analyze data, and much more.

- Aggregation pipelines can have one or more "**stages**". The order of these stages are important. Each stage acts upon the results of the previous stage.

Remember that the order of your stages matters. Each stage only acts upon the documents that previous stages provide.

- **$group:** groups documents by the unique _id expression provided. Don't confuse this _id expression with the _id ObjectId provided to each document.
- **$limit:** limits number of documents passed to next stage
- **$project:** only passes specified fields to next stage
- **$sort:** sorts all documents in the specified sort order

Aggregation Pipeline

- **$match:** This aggregation stage behaves like a find.
- It will filter documents that match the query provided.
- Using $match early in the pipeline can improve performance since it limits the number of documents the next stages must process.

Aggregation Pipeline

$addFields: This aggregation stage adds new fields to documents.

Example: this will return the documents along with a new field, avgGrade, which will contain the average of each restaurants grades.score.

```
db.restaurants.aggregate([
  {
    $addFields: {
      avgGrade: { $avg: "$grades.score" }
    }
  },
  {
    $project: {
      "name": 1,
      "avgGrade": 1
    }
  },
  {
    $limit: 5
  }
])
```

$addFields

```
{
  "name": “SC Cafe",
  "grades": [
    { "score": 80 },
    { "score": 85 },
    { "score": 90 }
  ]
}
```

```
{
  "name": “SC Cafe",
  "grades": [
    { "score": 80 },
    { "score": 85 },
    { "score": 90 }
  ],
  "avgGrade": 85
}
```

$project will show the name and avgGrade while the **$limit** will allow us to see only 5 documents.
What can **$sort** help us achieve here?

Aggregation Operators

- **$min**
- **$max**
- **$avg**
- **$sum**

Aggregation Pipeline

```
db.posts.aggregate([
  // Stage 1: Only find documents that have more than 1 like
  {
    $match: { likes: { $gt: 1 } }
  },
  // Stage 2: Group documents by category and sum each categories likes
  {
    $group: { _id: "$category", totalLikes: { $sum: "$likes" } }
  }
])
```

Purchase_orders Collection

```
{
  "_id": 4,
  "customer": "Alice Johnson",
  "product": "pizza",
  "total": 40,
  "orderDate": ISODate("2024-07-05T00:00:00Z"),
  "status": "completed"
}
```

Find the total amount of money spent by each customer


```
db.purchase_orders.aggregate(
    [
        {$match: {} },
        {$group: {_id: "$customer", totalSpend: { $sum: "$total"} } }
    ]
)
```

An empty **$match** does not filter anything but is often included for flexibility or as a placeholder to make the pipeline easier to update in the future. We can skip this completely if our pipeline does not require filtering.

Calculate the total amount spent by each customer for orders placed in 2024 and have status of Completed.

```
db.purchase_orders.aggregate([
  { $match: {
      orderDate: { $gte: ISODate("2024-01-01"), $lte: ISODate("2024-12-31") },
      status: "completed"
  }},
  { $group: {
      _id: "$customer", totalSpend: { $sum: "$total" }
  }}
])
```

Find how much has been spent on each product

```
db.purchase_orders.aggregate(
      [
            {$match: {} },
            {$group: {_id: "$product", totalRevenue: { $sum: "$total"}}}
       ]
)
```

Find how much has been spent on each product and sort it by price

```
db.purchase_orders.aggregate(
      [
            {$match: {} },
            {$group: {_id: "$product", totalRevenue: { $sum: "$total"}}},
            {$sort: {totalRevenue: -1}}
      ]
)
```

Find how much money each customer has spent on toothbrushes and pizza

```
db.purchase_orders.aggregate(
    [
        {$match: {product: {$in: ["toothbrush", "pizza"]} } },
        {$group: {_id: "$customer", totalSpend: { $sum: "$total"}}},
    ]
)
```

Filter with regex

```
db.purchase_orders.aggregate(
    [
        {$match: { product: { $regex: "^t"} } },
        {$group: {_id: "$product", totalRevenue: { $sum: "$total"}}},
    ]
)
```

Unwind

- Deconstructs an array field from the input documents to output a document for each element.
- Each output document is the input document with the value of the array field replaced by the element.
- *Flattening of the array*

Write a MongoDB query to find the average score for each cuisine.

```
db.restaurants.aggregate([
{ $unwind: "$grades" },
{
      $group: {
            _id: "$cuisine",
            avgScore: { $avg:
"$grades.score" }
      }
  }
])
```

The **$unwind** stage is used to break down the grades array and creates a separate document for each element in the grades array.

The **$group** stage is used to group the documents by the cuisine field and calculate the average score for each group.

```
[
 { "_id": 1, "name": "Restaurant A", "cuisine": "Italian", "grades": [ { "score": 90 }, { "score": 85 } ] },
 { "_id": 2, "name": "Restaurant B", "cuisine": "Italian", "grades": [ { "score": 80 }, { "score": 75 } ] },
 { "_id": 3, "name": "Restaurant C", "cuisine": "Chinese", "grades": [ { "score": 70 }, { "score": 95 } ] },
 { "_id": 4, "name": "Restaurant D", "cuisine": "Chinese", "grades": [ { "score": 80 }, { "score": 85 } ] }
]
```

UNWIND

```
[
 { "_id": 1, "name": "Restaurant A", "cuisine": "Italian", "grades": { "score": 90 } },
 { "_id": 1, "name": "Restaurant A", "cuisine": "Italian", "grades": { "score": 85 } },
 { "_id": 2, "name": "Restaurant B", "cuisine": "Italian", "grades": { "score": 80 } },
 { "_id": 2, "name": "Restaurant B", "cuisine": "Italian", "grades": { "score": 75 } },
 { "_id": 3, "name": "Restaurant C", "cuisine": "Chinese", "grades": { "score": 70 } },
 { "_id": 3, "name": "Restaurant C", "cuisine": "Chinese", "grades": { "score": 95 } },
 { "_id": 4, "name": "Restaurant D", "cuisine": "Chinese", "grades": { "score": 80 } },
 { "_id": 4, "name": "Restaurant D", "cuisine": "Chinese", "grades": { "score": 85 } }
]
```

```
{
  $group: {
    _id: "$cuisine",                          // Group by the 'cuisine' field
    avgScore: { $avg: "$grades.score" }       // Calculate the average score
  }
}
```

```
[
  { "_id": "Italian", "avgScore": 82.5 },
  { "_id": "Chinese", "avgScore": 82.5 }
]
```

Aggregation Pipeline

- **$count:** This aggregation stage counts the total amount of documents passed from the previous stage.

- **$lookup**: This aggregation stage performs a left outer join to a collection in the same database.

- **$out**: This aggregation stage writes the returned documents from the aggregation pipeline to a collection.

Count documents

$count: This aggregation stage counts the total amount of documents passed from the previous stage.

```
db.restaurants.aggregate([
  {$match: {cuisine: "Italian", likes: { $gt: 100 }}},
  {$count: "italianRestaurantCount"}
])
```

```
[
 { "_id": 1, "name": "Restaurant A", "cuisine": "Italian", "likes": 150 },
 { "_id": 2, "name": "Restaurant B", "cuisine": "Chinese", "likes": 120 },
 { "_id": 3, "name": "Restaurant C", "cuisine": "Italian", "likes": 80 },
 { "_id": 4, "name": "Restaurant D", "cuisine": "Italian", "likes": 200 },
 { "_id": 5, "name": "Restaurant E", "cuisine": "Mexican", "likes": 300 }
]
```

```
[
 { "italianRestaurantCount": 2 }
]
```

Count – group wise

```
db.restaurants.aggregate([
  {
    $group: {
      _id: "$cuisine",
      count: { $sum: 1 } //count can be any other field name of choice
    }
  }
])
```

```
[
 { "_id": 1, "name": "Restaurant A", "cuisine": "Italian", "likes": 150 },
 { "_id": 2, "name": "Restaurant B", "cuisine": "Chinese", "likes": 120 },
 { "_id": 3, "name": "Restaurant C", "cuisine": "Italian", "likes": 80 },
 { "_id": 4, "name": "Restaurant D", "cuisine": "Italian", "likes": 200 },
 { "_id": 5, "name": "Restaurant E", "cuisine": "Mexican", "likes": 300 }
]
```

```
[
 { "_id": "Italian", "count": 3 },
 { "_id": "Chinese", "count": 1 },
 { "_id": "Mexican", "count": 1 }
]
```

Counts for stages or groups

	$sum: 1 in $group	$count
Purpose	Count documents per group after grouping	Counts the total number of documents in the pipeline
Where to use	Inside $group stage	As last stage of pipeline

Orders example document

```
{
  "_id": 1,
  "customer": "John Doe",
  "items": [
    { "product": "Laptop", "price": 1200 },
    { "product": "Mouse", "price": 25 },
    { "product": "Keyboard", "price": 50 }
  ],
  "status": "Completed",
  "orderDate": "2024-10-10"
}
```

Calculate the total revenue generated by each product.

Practice Questions

1. Calculate the total revenue generated by each product.
2. Retrieve the 2 most recent completed orders.
3. How many orders are completed.
4. How many orders there are for each order status (e.g., "Completed", "Pending").

1. Retrieve the 2 most recent completed orders.

```
db.orders.aggregate([
  { $match: { status: "Completed" } },   // Filter for completed orders
  { $sort: { orderDate: -1 } },          // Sort by orderDate in descending order
  { $limit: 2 }                          // Limit the result to the 2 most recent orders
])
```

2. How many orders are completed.

```
db.orders.aggregate([
  { $match: { status: "Completed" } },
  { $count: "completedOrders" }
])
```

3. How many orders there are for each order status (e.g., "Completed", "Pending").

```
db.orders.aggregate([
  { $group: { _id: "$status", orderCount: { $sum: 1 } } }
])
```

4. Calculate the total revenue generated by each product.

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $group: { _id: "$items.product", totalRevenue: { $sum: "$items.price" } } }
])
```

Aggregation Pipeline Optimization

Optimization ensures queries run efficiently by reducing data processed at each stage.

Aggregation Pipeline Optimization

- **Place $match early**
  - Filter out unnecessary documents as soon as possible to reduce the number of documents passed to later stages.
  - Fewer documents to process = better performance.

```
[
  { "$sort": { "price": -1 } },
  { "$match": { "category": "electronics" } }
]
```

```
[
  { "$match": { "category": "electronics" } },
  { "$sort": { "price": -1 } }
]
```

Aggregation Pipeline Optimization

- **Use $project to reduce fields early**
  - Exclude unnecessary fields as soon as possible to reduce data size.
  - Smaller documents travel faster through the pipeline.

```
[
 { "$group": { "_id": "$category", "total": {
"$sum": "$price" } } }
]
```

```
[
 { "$project": { "category": 1, "price": 1 } },
 { "$group": { "_id": "$category", "total": {
"$sum": "$price" } } }
]
```

Aggregation Pipeline Optimization

- **Optimize $group**
  - Grouping can be expensive, so reduce the data size before grouping.
  - Smaller datasets group faster.

```
[
 { "$match": { "category": "electronics" } },
 { "$group": { "_id": "$brand", "averagePrice": { "$avg": "$price" } } }
]
```

Aggregation Pipeline Optimization

- **Index** Utilization
  - Ensure $match or $sort uses an appropriate index for faster execution.
  - Indexed operations are significantly faster.

Note: We will not be discussing the index implementation in this course.

Aggregation Pipeline Optimization

- **Avoid Unnecessary $unwind**
  - Use $unwind sparingly, and only when absolutely necessary.
  - Flattening arrays can be computationally expensive.

```
[
 { "$unwind": "$items" },
 { "$match": { "items.price": { "$gt": 100 } } }
]
```

```
[
 { "$match": { "items.price": { "$gt": 100 } } }
]
```

Aggregation Pipeline Optimization

- Combine stages where possible
- Use **$limit** and **$skip** efficiently
- Core idea is to reduce the number of documents passed to the next stage.

Aggregation Pipeline Optimization

- MongoDB automatically performs:
  - **Stage reordering**: Moves $match and $project earlier when safe.
  - **Index usage:** Uses available indexes in $match and $sort.