**Question 4 – MongoDB aggregation pipeline**

1. *Show how to use the MongoDB aggregation pipeline to query the data.*

**Using the MongoDB Aggregation Pipeline**

Is a sequence of stages, each taking the output of the previous stage as its input and transforming the document stream.  
Common stages include:

$match

* Filters documents (like **find()**)

$project

* Shapes each document by including, excluding, or computing fields.

EXAMPLE: all books by a given author

* **$match** filters documents whose authors array contains the given ObjectId.
* **$project** returns only the book’s title and isbn.

$unwind

* Deconstructs an array field into multiple documents.

$group

* Groups documents by a specified key and applies accumulator expressions **($sum, $avg,** etc.).

EXAMPLE: average page count by genre

* **$unwind** flattens the **genres** array.
* **$group** computes the average **page\_count** for each genre.

$lookup

* Performs a left-outer join with another collection.

EXAMPLE: merges each **order** with its matching **customer** document

$graphLookup

* Recursively joins documents (useful for tree- or graph-structured data).

EXAMPLE: for each employee, it will recursively follow their **manager\_id** links up the chain

* Hany for tree-pattern queries

1. *Compare selected queries from your 3 assignments.*

**Comparing Selected Queries Across Assignment**

**SQL (Assignment 1)**

Books by author

* **SELECT b.title, b.isbn  
  FROM books b  
  JOIN book\_authors ba ON b.id = ba.book\_id  
  WHERE ba.author\_id = ?;**

Average page count by genre

* **SELECT g.genre, AVG(b.page\_count) AS avg\_page\_count  
  FROM books b  
  JOIN book\_genres g ON b.id = g.book\_id  
  GROUP BY g.genre;**

Total sales per customer

* **SELECT customer\_id, SUM(total\_price) AS total\_sales  
  FROM orders  
  WHERE customer\_id = ?  
  GROUP BY customer\_id;**

**Document (Assignment 2)**

Books by author

* **db.books.find({ authors: ObjectId("…") });**

or, as a pipeline:

* **[$match:{authors:…}, $project:{title:1,isbn:1,\_id:0}]**

Average page count by genre

* **[$unwind:"$genres", $group:{\_id:"$genres",avg\_page\_count:{$avg:"$page\_count"}}, $project:{genre:"$\_id",avg\_page\_count:1,\_id:0}]**

Total sales per customer

* **[$match:{customer\_id:ObjectId("…")}, $group:{\_id:"$customer\_id",total\_sales:{$sum:"$total\_price"}}, $project:{customer\_id:"$\_id",total\_sales:1,\_id:0}]**

**Graph (Assignment 3)**

Books by author

* **query($id:ID!){  
   author(where:{id:$id}){  
   books{title,isbn}  
   }  
  }**

Average page count by genre

Via a custom resolver/@cypher:

* **type Query {  
   avgPageCountByGenre: [GenreAvg]! @cypher(statement:"MATCH (b:Book)-  
   [:HAS\_GENRE]->(g) RETURN g.name AS genre, AVG(b.page\_count) AS   
   avg\_page\_count")  
  }**

Total sales per customer

* **query($id:ID!){  
   customer(where:{id:$id}){  
   ordersAggregate{  
   sum{total\_price}  
   }  
   }  
  }**

Each column shows the idiomatic way of expressing the same data-retrieval requirement in:

* **Relational SQL** (Assignment 1),
* **MongoDB’s query API** and **aggregation pipeline** (Assignment 2), and
* **GraphQL/Neo4j** (Assignment 3).

This highlights how the aggregation pipeline replaces JOIN + GROUP BY in SQL with a sequence of stages ($lookup, $unwind, $group, etc.), and how in GraphQL/Neo4j you often rely on automatic resolvers or custom @cypher directives to achieve the same end.