* **MongoDB:-**
→ It derived from the humongous (huge, giant).
→ MongoDB is the most babular and trending database.

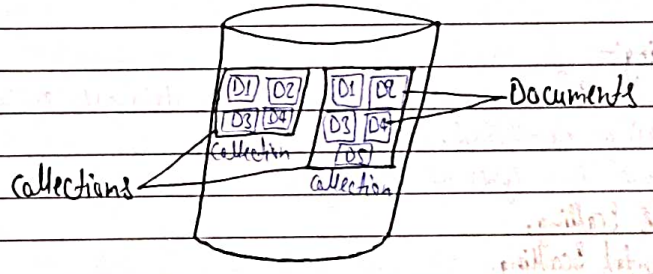
* **Where can we use MongoDB?:-**
→ We can use MongoDB everywhere.
→ We can use it to develop desktop application, mobile application and web application.

* **Stack:-**
→ The technologies which are used to develope web app is known as stack.
M:- MongoDB
E:- ExpressJS [It is use to develop the backend.]
A:- Angular /R:- ReactJS [these are the frameworks and libraries for the frontend.]
N:- NodeJS (To provide server site environment.)

* **What is the type of MongoDB database?:-**
→ This is a document oriented database. It is No-SQL database.
→ **No-SQL:-** ["Non-SQL" or "Non-RDBMS"]:-
→ Any database which do not follows relational database or SQL is known as a Non-SQL database.

* **RDBMS V/S No-SQL:-**

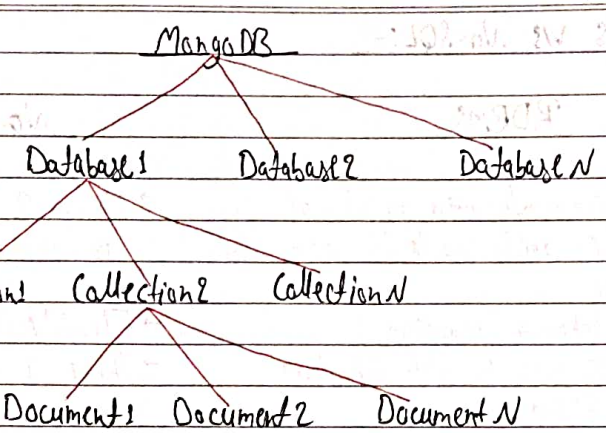
RDBMS	No-SQL
→ Most commonly used relational database are:- Oracle SQL or My-SQL	→ MongoDB is the most commonly used non-SQL database.
→ fixed schema (Structure)	→ Flexible schema (documents)
→ We use joins to fetch the data from RDBMS.	→ There is no concept of joins in mongodb.



MongoDB

* **Note:-** Data will be stored in separate documents in MongoDB. Each document is independent.

* **MongoDB Structure:-**
→ MongoDB physical database contains logical databases.
→ Each database contains several collection. Collection is like a table in RDBMS
→ Each collection contains several documents. Document is like a record (row) in RDBMS.



* **Scalling:-**
  → Scalling is ability of a database system to handle increasing workload.
→ We have two types of scalling:-
(a) Verticle Scalling,
(b) Horizontal Scalling.

* **Verticle Scalling:-**
  → Verticle scalling involves increasing the capacity of a single server in a database-system.
→ This can be achieved by upgrading the hardware components such as increasing the RAM, capacity of CPU.

* **Horizontal Scalling:-**
  → Horizontal scaling involves increasing the capacity by more servers or distributing work load to other machine.
* **Note:-** Relational Database only support verticle scalling and No-SQL databases supports both verticle and horizontle scalling.

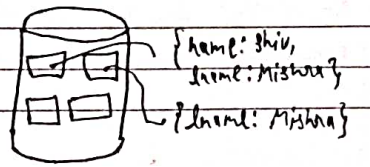
* **Types of No-SQL Databases:-**
  → Document Type Database:- (MongoDB, CouchDB)
→ Key-Value Store Database:- (Redis, DynamoDB)
→ Columns Oriented Database:- (Cassendra, BigTable)
→ Graph Database:- (Neo4J, GraphDB)

* **MongoDB:-**
  → MongoDB is a cross-platform, document oriented database that provides high availability and scalling, MongoDB works on the concept of collection and document.

* **Database:-**
  → Database is a physical container for collection.
→ Each database gets it's own set of collections.
→ A single MongoDB server as typically multible databases.

* **Collections:-**
  → Collection is a group MongoDB documents. It is an equivalent to RDBMS table.
→ In mongoDB, we can store our data in the form of JSON, BSON, XML, Documents.
→ A collection exists within a single database.

* **Documents:-**
  → A document is a set of key-value pairs and documents have dynamic fields. Dynamic schema ma means that documents in the same collection, do not follows or do not need to have the same set of fields/structure.



* JSON:-[Java Script Object Notation]:-

→ JSON is a light weight format used to exchange and store data.

→ It is designed to be easy for humans and machine to under.

→ It is JSON uses a simple structure of key-value pair.

Data in JSON is enclosed within curly braces ({}) and consist of key-value pairs.

* Datatybes of JSON:-

→ Number,
→ String,
→ Boolean,
→ NULL,
→ Object,
→ Array.

* JSON Syntax:-

→ Keys and Strings must be enclosed in double quotes (" ") ["key": "value"]

→ Commas (,) are used to separate key/value pairs and elements with arrays.

→ JSON must always be enclosed in curly braces. Each key is follow by a colon (:) and it's corresponding value.

* How will we store dates in JSON?:-

→ When including dates in JSON, it is generally recommended to use the ISO8601 standard format.

→ The ISO8601 format represents date and times in a structured and unambiguous manner.

* Format:-

1→ "YYYY-MM-DD":- "2023-06-25"

2→ "YYYY-MM-DDTHH:MM:SS":- "2023-06-25T18:27:00"

* Example of JSON:-

→
```
{ "empno": 7369,
  "ename": "SMITH",
  "JOB": "CLERK",
  "MGR": 7902,
  "Hiredate": "1980-12-17",
  "SAL": 800,
  "Comm": NULL,
  "Deptdetails": { "deptno": 20,
                   "DNAME": "RESEARCH",
                   "LOC": "DALLAS"}
}
```

* How to create database in MongoDB?:-

→

Syntax:- "USE Database-Name"

Ex:- use Mongodbclass

* How to create a collection in a Database:-

Syntax:- " db.createCollection("collection-name")"

Ex:- db.createCollection("developers")

* How to display all the collections present in a database?:-

Syntax:- "show collections" OR "show tables"

* How to delete a collection from database?:-

Syntax:- " db.collectionName.drop()"

Ex:- db.developers.drop()

* How to delete or drop a database?:-
Syntax:- "db.dropDatabase()"

* How to insert a document in MongoDB collection?:-
→ We will use insert one function ('insertOne()'), or 'insertMany()'.
Syntax:-
"db.collectionName.insertOne()"
Ex:-
db.testing.insertOne({"name": "Binod"})

* How to display the documents present in collection?:-
Syntax:- "db.collectionName.find()"
Ex:- db.testing.find()

InsertMany:-
→ It is use to insert multiple document in a collection at a time.
Syntax:- "db.collectionName.insertMany()"

* Write a query to display details of oppoF19 phone from mobiles collection.
Query:- db.mobiles.find({title: 'OPPOF19'})

* ~~RDBMS~~ RDBMS where clause equivalent in MongoDB:-
:- Equivalent
Operator:- (:) → equals to.
2→ Greater Than:- ($gt)
3→ Less Than:- ($lt)
4→ Greater than equals to:- ($gte)
5→ Less than equals to:- ($lte)
6→ Not equals to:- ($ne)
7→ In operator:- ($in)
8→ Not In operator:- ($nin)

* Note:- Embedded Document/Nested JSON:-
A document (JSON) inside another document (JSON) is known as an embedded document or nested JSON.

* Write a command to display the products whose rating is greater than 4 from store collection.
Query:- db.store.find({"rating.rate": {$gt: 4}})

* WACTD the products whose price is greater than 900.
Query:- db.store.find({price: {$gt: 900}})

* WACTD products whose rating is less than 3.
Query:- db.store.find({"rating.rate": {$lt: 3}})

* WACTD the products whose id is 12,13,14.
Query:- db.store.find({id: {$in: [12,13,4]}})

* Projection:-
Syntax:- find({QUERY}, {PROJECTION})
mandatory, optional
Ex:- db.customers.find({}, {name:1}) OR db.customers.find({}, {_id:0})

* WACTD name of the customers whose age is greater than 25.

Query:- db.customers.find({age:{$gt:25}},{name:1})

* Find One():-

Syntax:- db.collectionName.findOne()

→ It is used to display a single document or first document in collection.

Ex:- db.customers.findOne()

* WACTD name of the customers who are living in state CA.

Query:- db.customers.find({"address.state":"CA"},{name:1,_id:0})

* WACTD name and age of the customers who are 25, 29, 32 years old.

Query:- db.customers.find({age:{$in:[25,29,32]}},{_id:0,name:1,age:1})

* WACTD name and state and city for the customers who doesn't live in CA and NY states.

Query:- db.customers.find({"address.state":{$nin:["CA","NY"]}},{_id:0,name:1,"address.state":1,"address.city":1})

* WACTD name, age and email for the customers whose zipcode is 54321 or 67890.

Query:- db.customers.find({"address.zip":{$in:["54321","67890"]}},{_id:0,name:1,age:1,email:1})

* Logical Operators:-

(i) And → $and
(ii) OR → $or
(iii) NOT → $not
(iv) NOR → $nor

And:- To do the commands based on and's operator we need to use ($and) operator.

Syntax:- {$and:[{k:v},{k:v}]}

* WACTD name and ~~interest~~ age for all the customers whose age is greater than 25 but less than 30.

Query:- db.customers.find({age:[{$gt:25},{$lt:30}]},{_id:0,name:1,age:1}) ××

✓✓ db.customers.find({$and:[{age:{$gt:25}},{age:{$lt:30}}]},{_id:0,name:1,age:1})

* WACTD name and interest of the customers whose interests are movies, music and cooking. Show only those customers who have all three interests.

Query:- db.customers.find({$and:[{interests:"movies"},{interests:"music"},{interests:"cooking"}]},{_id:0,name:1,interests:1})

* WACTD name, age and city for the customers if they lives in any city or there age is greater than 30.

Query:- db.customers.find({$or:[{"address.city":"ANYCITY"},{age:{$gt:30}}]},{_id:0,name:1,age:1,"address.city":1})

* WACTD name, age, state along with interests for the customers who ages are 25 or 26, 28 or, who lives in state "CA" or "NY" and having interests in sports, yoga, cooking.

Query:- db.customers.find({$and:[{age:{$in:[25,26,28]}},{"address.state":{$in:["CA","NY"]}},{interests:{$in:["sports","yoga","cooking"]}},{_id:0,name:1,age:1,"address.state":1,interests:1}])

* NOR Operator:-

* WACTD name and interests for all the customers who are not interested in music or cooking or yoga.

Query:- db.customers.find({$nor:[{interests:"music"},{interests:"cooking"},{interests:"yoga"}]},{_id:0,name:1,interests:1})

Query:- WACTD all the details of customers whoose age is not less than 25 and they should not be interested in music or movies and they should not living in state 'TX'.

Query:- db.customers.find({$nor:[{age:{$lt:25}},{interests:{$in:['music','movies']}},{"address.state":"TX"}]})

* UPDATES OPERATORS:-
  - → $set
  - → $unset
  - → $rename
  - → $[<identifiers>]

* $set:-
  - → The $set operator replaces the value of a field with the specified value.
  - → The set operator expression has the following format:-

{$set:{<field>:<values>,....}}

* $unset:-
  - → The $unset operator deletes the particular fields.

{$unset:{<field>:"",...}}

* $rename:-
  - → The $rename operator updates the field.

{{$rename:{<field>:<newname>,...}}

  - → The new field name must differ from the existing field name.
  - → To specify a field in an embedded document, use (.) dot notation
    eg:- "address.state"

* $[<identifier>]:-
  - → It acts as a placeholder to update all the elements that match the arrayFilters condition for the document that match the query condition.

* updateOne(filters, update, options):-
  - → Update single of document within the collection based on the filter.
  - → Here options is optional.

* options:-
  - ✓ upsert:- It's datatype is boolean.
  - → When true, updateOne() either:-
  - (a) (b) Creates a new document if no documents match the filter.

* WACTUpdate the name to Harsh from Rahul.

Query:- db.student.updateOne({name:"rahul"},{$set:{name:"harsh"}})

* WACT remove number field from the document.

Query:- db.student.updateOne({},{$unset:{number:""}})

* WACT update name and number fields in all the documents.

Query:- db.students.updateMany({},{$rename:{name:'naam',number:'contact'}})

* UPSERT:-

* **Delete :-**

It is use to delete a document (deleteOne()) or use to delete multiple documents (deleteMany()), which satisfies the filter condition.

Syntax :- deleteOne (<filter>)
deleteMany (<filter>)

* WACT delete a document whose id is one.

Query :- db.students.deleteOne ({_id:1})

* WACT delete all the documents where address is patel.

Query :- db.students.deleteMany ({address: 'patel'})

* **Array-Filters :-**

→ An array of filter documents that determines which array elements to modify for an update operation on an array field.

→ It is always used with $[] (identifiers)

* WACT update 'sql' to "mongodb" in tech skills array.

Query :- db.students.updateMany ({}, {$set: {"skills.techskills.$[skill]": "mongodb"}}, {arrayFilters: [{skill: 'sql'}]})

* **MongoDB Limit Records :-**

 # Limit :-

→ To limit the records in mongoDB, we need to use limit() method.

→ This method accepts one number type argument, which is the number of documents that we want to be display.

Syntax :- db.collectionName.find().limit(number)

 # skip() :- Apart from limit(), skip() also accepts number type argument. And is used to skip the number of documents.

Syntax :- db.collectionName.find().limit().skip(number).

* **MongoDB Sort Records :-**

 # sort() :- For sorting the documents we need to use sort method. This method accepts a document containing a field alongwith there sorting orders.

→ To specify sorting order we will use one or -one (-1).

→ '1' :- It means ascending order,
-1 :- It means descending order.

Syntax :- db.collectionName.find().sort ({field: order})

ex :- db.customers.find ({}, {_id:0, name:1, age:1}).sort({age:1})

 # count() :- This will return the count of the documents present in a collection or it will be based upon the filter conditions that we will give in find().

Syntax :- db.collectionName.find().count()

* WACTD name and age of all customers who are older than 30 and sort them on the basis of age in descending order.

Query :- db.customers.find({age: {$gt: 30}}, {_id:0, name:1, age:1}).sort({age:-1})

* WACTD number of customers who lives in state 'CA' and are interested in 'music', 'movies', 'yoga'.

Query :- db.customers.find({$and: [{"address.state": 'CA'}, {interests: {$in: ['music', 'movies', 'yoga']}}]}).count()

* explain():- It will be always used after find() method.
→ The explain() is used to retrieve information about the execution and performance, statistics of query.

* WAQT find persons with the age greater than 30:-
Query:- db.persons.find({age:{$gt:30}}).explain()
db.persons.find({age:{$gt:30}}).explain('executionStats')

→ indexes support the efficient resolution of queries. Without indexes, mongoDB must scan every document of a collection to select those ~~statement~~ documents that matches the filter condition.
→ This scan is highly in-efficient and requires mongoDB to process a large amount of data.

* createIndex():- To create an index we need to use createIndex() method.
Syntax:- db.collectionName.createIndex({key:1})

→ Here key is the name field on which you want to create index and '1' is for ascending and '-1' is for descending.
Ex:- db.persons.createIndex({age:1}) // age-1:

* getIndex():- It is use to display all the indexes present in a collection.

* dropIndex():- It is use to drop or delete the index.
Syntax:- db.collectionName.~~get~~dropIndex()
Ex:- db.persons.dropIndex('age_1')

* WAQT) persons who are female.
Query:- db.persons.find({gender:'Female'}).count()

* Compound Index:-
→ Whenever we create index of more than one field then it is known as compound index.
Ex:- db.persons.createIndex({gender:1, age:1})
db.
→ In compound index the order of fields matters.

* Unique Index:- db.persons.createIndex({name:1},{unique:true})
→ This means that in name fields every value is unique and we can not insert any repeated name.

* Text Index:- This index allows for text search on all fields with string content.
#$text operator:- It is an operator in mongoDB used for text search. It allows us to perform full text searches on a particular field (String type) within a collection.
→ To use $text operator on a particular field, there should be a text ~~field~~ index on that field.

#$search operator:- It is use within $text operator.
→ It specifies the search term or set of words who search within the index text field.
→ When combined with text operator it enables us to search for documents containing the specified text.

* Note:- $text is the main operator that enables text search, while search is used within $text to define the specific search term or words

Syntax:- createIndex({<field>: "text"})
Ex:- db.class.createIndex({desc: "text"})

* WAQTD documents in which the 'youtuber' is mentiond in desc.
Query:- db.class.find({$text: {$search: 'youtuber'}})

* To make this search case-sensitive:-
db.class.find({$text: {$search: 'YOUTUBER', $caseSensitive: true}})

* multiple search:-
db.class.find({$text: {$search: 'youtuber prince'}})
=> this will work as or operator.
* WAQTD the documents in which the youtuber or prince mentioned.

* Note:- We can create only one text index in a collection.

* Foreach:-
* Pattern Matching in MongoDB:-
Regular Expression:-
→ In mongoDB, A regular expression is a powerful tool used for pattern matching within 'string fields'.
→ RE allows us to search for strings based on specific patterns.
→ RE provides a flexible way to search text in mongoDB.
→ RE in mongoDB are typically used with '$regex' operator, which allows us to specifies the pattern to be matched.
Syntax:- {<field>: {$regex: /pattern/}}

/pattern/ - The pattern is enclosed between forward slashes.
Start with:- ^ Ex:- /^.{}/ ends with:- $ case-insensitive:- i having-char:- ||
* WAQTD names which starts with character 'W'.
Query:- db.persons.find({name: {$regex: /^W/}})

* To make it case-insensitive:-
→ For that we use 'i' in the last of pattern to make it case-insensitive.
Ex:- db.persons.find({name: {$regex: /^W/i}})

* WAQTD documents where names ends with 'L'.
Query:- db.persons.find({name: {$regex: /L$/}})

* Note:- $regex is not mandatory for pattern matching we can also write like that:-
db.persons.find({name: /^w/i})

* WAQTD documents in which names are having character 'A'.
Query:- db.persons.find({name: /a/i})

* WAQTD names which starts with 'M' or 'A'.
Query:- db.persons.find({$or: [{name: {$regex: /^M/i}}, {name: /^A/}]})

* WAQTD documents in which 'content' and 'youtuber' is mentioned.
Query:- db.ber.db.class.find({$and: [{desc: /content/i}, {desc: /youtuber/i}]})

* WAQTD documents where names starts with 'G' and ends with 'e':

Query:- db.persons.find({$and:[{name:/^G/i},{name:/e$/i}]})

* WAQTD documents in which name has character 'j' as the third character:

Query:- db.persons.find({name:/^..j/i})

* WAQTD documents in which names are having character 'n' as the third last character of there names.

Query:- db.persons.find({name:/n..$/i})

* Aggregation in MongoDB:-

→ Aggregation in MongoDB is the process of processing, grouping and transforming data from multiple documents within a collection.

→ It allows us to perform complex data manipulations on our data.

→ MongoDB's aggregation framework provide a set of powerful tools to perform task like filtering, grouping, sorting data.

→ It is like the SQL group by clause but offers a more flexibility.

→ To perform aggregation operations:-

we can use:- aggregation

(a) Aggregation Pipelines:- An aggregation pipelines consist of one or more stages that process documents. Each stage performs an operations on multiple documents for example, A stage can filter document, A stage can create groups of the documents and calculate values.

→ The documents that are output from a stage are passed to the next stage.

→ An aggregation pipeline can returns results for group of document for example:- return the total, average, minimum and maximum values.

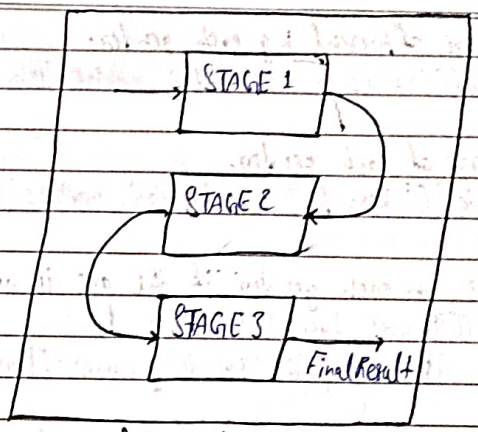


fig:- Pipeline

→ To achieve aggregation in MongoDB we will use aggregate() method

First Stage:-

→ $match

Note:- In aggregate() function we will pass array of different stages.

Ex:- db.persons.aggregate([{$match:{age:{$gt:30}}}])

Second Stage:-

→ $group

Ex:- db.persons.aggregate([{$group:{_id:"gender"}}])

→ _id:- we will write name of field on which we want to create groups.

* WAQTD number of persons present in each gender.

Query:- db.persons.aggregate([{$group:{_id:"gender", totalPersons:{$sum:1}}}])

* WAQTD number of gender persons in each gender whose ages are greater than 27.

Query:- db.persons.aggregate([{$match:{age:{$gt:27}}},{$group:{_id:"$gender", totalPersons:{$sum:1}}}])

* WAQTD minimum age of present in each gender.

Query:- db.persons.aggregate([{$group:{_id:"$gender", minAge:{$min:"$age"}}}])

* WAQTD average age of each gender.

Query:- db.persons.aggregate([{$group:{_id:"$gender", avgAge:{$avg:"$age"}}}])

* WAQTD documents in each gender if the age is greater than 20.

Query:- db.people.aggregate([{$match:{age:{$gt:20}}},
{$group:{_id:"$gender", names:{$push:"$name"}}}])

* Note:- If we want to display whole document inside $push operator, we have to write {$push:"$$ROOT"}

* WAQTD hobbies for all the people in each age group and the age should be grunter than 30.

Query:- db.people.aggregate([{$match:{age:{$gt:30}}},
{$group:{_id:"$age", hobbies:{$push:"$hobbies"}}}])

Third Stage:-

→ $unwind:- It is use to ~~remove~~ unpack the fields which we will give.

Ex:- db.people.aggregate([{$unwind:"$hobbies"},
{$match:{age:{$gt:30}}},
{$group:{_id:"$age", hobbies:{$push:"$hobbies"}}}])

* WAQTD names of the people in each age groups.

Query:- db.people.aggregate([{$group:{_id:"$age", name:{$push:"$name"}}}])

* Forth Stage:-

→ $sort

* WAQTD ages (group) in which there are atleast two people.

Query:- db.people.aggregate([{$group:{_id:"$age", count:{$sum:1}}},
{$match:{count:{$gte:2}}}])

OR

db.people.aggregate([{$group:{_id:"$age", count:{$sum:1}}},
{$match:{count:{$gte:2}}},
{$sort:{_id:1}}])

* Fifth Stage:-

→ $skip

→ $limit

* Operator:-

→ $push

→ $max

→ $min

→ $avg

→ $sum

→ $multiply [" ", " "]

→ $divide [" ", " "]

→ $round

Sixth Stage

→ The $project stage in mongoDB's aggregation framework is used to reshape documents by specifying which fields to include or exclude.

We can create new fields, rename existing fields, and apply expressions

to manipulate data.

* WAQTD name and age for all the people but age should be double of their age.

Query:- db.people.aggregate([{$project:{_id:0, name:"$name", ageNew:{$multiply:["$age",2]}}}])

* WAQTD average age in each hobbies.

Query:- db.people.aggregate([{$unwind:"$hobbies"}, {$group:{_id:"$hobbies", avgAge:{$avg:"$age"}}}])

db.people.aggregate([{$unwind:"$hobbies"}, {$group:{_id:"$hobbies", avgAge:{$avg:"$age"}}}, {$project:{_id:0, hobby:"$_id", age:{$round:"$avgAge"}}}])

* Next Stage:-

_id	custName	email	proDetails
101	John	J@	[ ]
102	Jane	Ja@	[ ]
104	BOB	B@	

_id	proName	custID
1	Smartphone	101
2	LabTop	102
3	Headphones	103

→ $lookup:-

→ In mongoDB the lookup stage is an aggregation pipeline stage used to perform a left outer join between documents from two collections.

→ It allows us to combine documents from one collection with document from another collection based on a specified field that appear in both collections.

Syntax:-

{$lookup:
{from: <collection to join>,
localField: <field from the input documents>,
foreignField: <field from the document of the from "from" collection>,
as: <output array field>
}
}

Ex:- db.customers.aggregate([{$lookup:
{from: "products",
localField: "_id",
foreignField: "custId",
as: "proDetails"}
}])

HW

Q1)

Query:- db.people.aggregate([{$group:{_id:{gender:"$gender", city:"$city"}, count:{$sum:1}}}])

Eg)

Query:- db.people.aggregate([{$unwind:"$hobbies"}, {$group:{_id:"$hobbies", minAge:{$min:"$age"}, maxAge:{$max:"$age"}}}])