MONGODB

INTRODUCTION

- MongoDB is a popular NoSQL (Not Only SQL) database management system.
- It falls under the category of document-oriented databases.
- MongoDB stores data in flexible, JSON(Java Script Object Notation-like documents, called BSON.
- No fixed schema: Each document in a collection can have a different structure.
- Supports horizontal scalability and high availability.

- MongoDB was developed by 10gen (now MongoDB, Inc.) and released in 2009 as an open-source project.
- It is written in C++ and widely used in modern web and mobile application development.
- JSON is an open std file format that uses human readable text to store and transmit.

Why MongoDB?

- Ideal for handling unstructured or semi-structured data.
- Scalable for handling large datasets and high traffic loads.
- Rapid development and agile changes due to schema flexibility.
- Rich query capabilities for complex data retrieval.

- MongoDB is used in various industries, including e-commerce, social media, finance, and healthcare.
- Commonly used for content management, catalog systems, real-time analytics, and IoT applications.
- MongoDB has a thriving community and a rich ecosystem of tools and libraries.
- Offers both a free, open-source version and a commercial version with additional features.

Why MongoDB-JSON

- MongoDB natively stores data in a format similar to JSON called BSON (Binary JSON).
- BSON extends JSON to include additional data types like dates and binary data.
- JSON documents in MongoDB are schema-less, which means you can insert documents into a collection without needing to
- JSON allows you to represent complex data structures with nested arrays and objects.define a rigid, predefined schema.

- JSON is human-readable, making it easy to work with, debug, and understand, both for developers and administrators.
- This readability aids in troubleshooting and data exploration.

- **Client-Side JSON**: When a client application interacts with MongoDB, it typically sends data in JSON format. For example, when inserting a new document into a collection, the client sends a JSON document as the input.
- **Server-Side BSON**: MongoDB's server-side components, including the storage engine and query engine, work with BSON internally. So, when the JSON data arrives at the MongoDB server, it's automatically converted from JSON to BSON before being processed or stored.

- **Insertion**: When you insert a document using the MongoDB driver or API, the driver performs the conversion from JSON to BSON before sending the data to the server. The server then stores the BSON document in the appropriate collection.

- **Query Execution**: When you query MongoDB, you typically specify query conditions in JSON format. These queries are also automatically converted from JSON to BSON by the MongoDB query engine before execution.
- **Response to Clients**: When the server retrieves data from the database in response to a query, it sends the result back to the client in BSON format. The client-side MongoDB driver then deserializes the BSON data into JSON for the application to use.

- **Network Communication**: BSON is also used for efficient network communication between the MongoDB server and client applications. Data is serialized into BSON format when sent over the network and deserialized on the receiving end.

The JSON code can as follows

```
 {
FirstName:John,
LastName:Mathews,
ContactNo: [+12345678900, +12344445555]
}
{
FirstName: Andrews,
LastName: Symmonds,
ContactNo: [+4567890 1234, +45666667777]
}
```

{

First Name Mable,

LastName: Mathews,

ContactNo: +789 12345678

}

Creating or Generating Unique Key

- Each JSON document should have a unique identifier.
- It is the _id key.
- It is similar to the primary key in relational databases.
- This facilitates search for documents based on the unique identifier.
- An index is automatically built on the unique identifier.

Database

- It is a collection of collections.
- In other words, it is like a container for collections.
- It gets created the time that your collection makes a reference to it.
- This can also be created on demand.
- Each database gets its own set of files on the file system.
- A single MongoDB server can house several databases.

Collection:

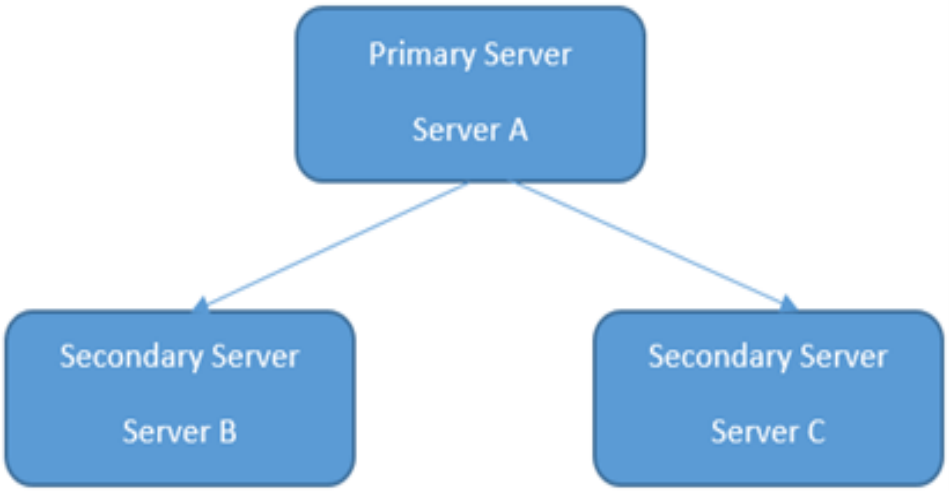
- A collection is created on demand.
- It gets created the first time that you attempt to save a document that references it.
- A collection exists within a single database.
- A collection holds several MongoDB documents.
- A collection does not enforce a schema.
- This implies that documents within a collection can have different fields.
- Even if the documents within a collection have fields, the order of the fields can be different

Document

- A document is analogous to a row/record/tuple in an RDBMS table.
- A document has a dynamic schema.

Replication

- Replication is referred to the process of ensuring that the same data is available on more than one Mongo DB Server.
- Replication provides data redundancy and high availability.
- It helps to recover from hardware failure and service interruptions.
- In MongoDB, the replica set has a single primary and several secondaries.
- Each write request from the client is directed to the primary.



Replication happens from the Primary to secondary servers

- **Adding a Secondary using rs.add()**
- rs.add("ServerB")
- rs.add("ServerC")

To remove replica server

rs.remove("ServerC")

Sharding

- It means that the large dataset is divided and distributed over multiple servers or shards.
- Each shard is an independent database and collectively they would constitute a logical database.
- The prime advantages of Sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage.
   For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house 256 GB data.
2. Sharding reduces the number of operations that each shard handles.
   For example, if we were to insert data, the application needs to access only that shard which houses that data.

Create Database

- MongoDB use DATABASE_NAME is used to create database.
- The command will create a new database if it doesn't exist, otherwise it will return the existing database.
- Example:
- use db1;
- Note: In MongoDB, a database is not actually created until it gets content.

Create Collection

Method 1:

- You can create a collection using the createCollection() database method.

Example: db.createCollection(“students")

{ ok: 1 }

Method 2:

- create a collection during the insert process.

Example:

db.posts.insertOne({“Name": "Priya"})

Insert Documents

Method 1:

- To insert a single document, use the insertOne() method.
- This method inserts a single object into the database.

```
db.posts.insertOne
({name : “achu",
class: “mca.",
category: "News",
likes: 1,
tags: ["news", "events"],
date: Date() })
```

MongoDB limit() Method

- In MongoDB, limit() method is used to limit the fields of document that you want to show.
- Sometimes, you have a lot of fields in collection of your database and have to retrieve only 1 or 2. In such case, limit() method is used.
- The MongoDB limit() method is used with find() method.
- Syntax:
  db.COLLECTION_NAME.find().limit(NUMBER)

```
{
Course: "Java", details: { Duration: "6 months", Trainer:
   "Sonoo Jaiswal" }, Batch: [ { size: "Medium", qty: 25 } ],
   category: "Programming Language" },
 {
Course: ".Net", details: { Duration: "6 months", Trainer:
   "Prashant Verma" }, Batch: [ { size: "Small", qty: 5 }, { size:
   "Medium", qty: 10 }, ], category: "Programming Language"
   },
{
 Course: "Web Designing", details: { Duration: "3 months",
   Trainer: "Rashmi Desai" }, Batch: [ { size: "Small", qty: 5 }, {
   size: "Large", qty: 10 } ], category: "Programming Language"
   } ];
```

- db.javatpoint.find().limit(1)
- After the execution, you will get the following result Output:
- { "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java", "details" : { "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [{ "size" : "Medium", "qty" : 25 }], "category" : "Programming Language" }

MongoDB skip() method

- In MongoDB, skip() method is used to skip the document.
- It is used with find() and limit() methods.
- Syntax
  db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)

{
Course: "Java", details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" }, Batch: [{ size: "Medium", qty: 25 }], category: "Programming Language" },
{
Course: ".Net", details: { Duration: "6 months", Trainer: "Prashant Verma" }, Batch: [{ size: "Small", qty: 5 }, { size: "Medium", qty: 10 },], category: "Programming Language" },
{
Course: "Web Designing", details: { Duration: "3 months", Trainer: "Rashmi Desai" }, Batch: [{ size: "Small", qty: 5 }, { size: "Large", qty: 10 }], category: "Programming Language" }

- db.javatpoint.find().limit(1).skip(2)
- After the execution, you will get the following result Output:
- { "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing", "det ails" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" : [{ " size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 }], "category" : " Programming Language" }

MongoDB sort() method

- In MongoDB, sort() method is used to sort the documents in the collection.
- This method accepts a document containing list of fields along with their sorting order.
- The sorting order is specified as 1 or -1.
- 1 is used for ascending order sorting.
- -1 is used for descending order sorting.

Select All Documents in a Collection

- To select all documents in the collection, pass an empty document as the query filter parameter to the find method.
- The query filter parameter determines the select criteria:
- db.inventory.find({})

- The following example selects from the inventory collection all documents where the status equals "D":
- db.inventory.find({ status: "D" })
- This operation corresponds to the following SQL statement:
- SELECT * FROM inventory WHERE status = "D"

Specify Conditions Using Query Operators

- A query filter document can use the query operators to specify conditions in the following form:
- {<field1> : {<operator1> :<value1> }, ... }
- The following example retrieves all documents from the inventory collection where status equals either "A" or "D":
- db.inventory.find({ status: { $in: ["A", "D"] } })

- Note: Although you can express this query using the **$or** operator, use the $in operator rather than the $or operator when performing equality checks on the same field.
- The operation corresponds to the following SQL statement:
- SELECT * FROM inventory WHERE status in ("A", "D")

Specify AND Conditions

- A compound query can specify conditions for more than one field in the collection's documents.
- Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

- The following example retrieves all documents in the inventory collection where the status equals "A" and qty is less than ($lt) 30:
- db.inventory.find({ status: "A", qty: { $lt: 30 } })
- The operation corresponds to the following SQL statement:
- SELECT * FROM inventory WHERE status = "A" AND qty < 30

Specify OR Conditions:

- Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.
- The following example retrieves all documents in the collection where the status equals "A" or qty is less than ($lt) 30:
- db.inventory.find({ $or: [{ status: "A" }, { qty: { $lt: 30 } }] })
- The operation corresponds to the following SQL statement: SELECT * FROM inventory WHERE status = "A" OR qty < 30

Specify AND as well as OR Conditions:

- In the following example, the compound query document selects all documents in the collection where the status equals "A" and either qty is less than ($lt) 30 or item starts with the character p:
- db.inventory.find({ status: "A", $or: [{ qty: { $lt: 30 } }, { item: /^p/ }] })
- The operation corresponds to the following SQL statement:
- SELECT * FROM inventory WHERE status = "A" AND (qty < 30 OR item LIKE "p%")

MongoDB Query Language with Arrays

- An array is a data type used to store an ordered list of values within a single document.
- Arrays can contain a mix of different data types, and they are often used to represent collections of related or similar values.
- **Structure**: Arrays in MongoDB are embedded within documents as fields. These fields can contain one or more values, making them an ordered list.

- **Data Types**: Arrays can store values of various data types, including strings, numbers, subdocuments (nested documents), other arrays, and even binary data.
- **Access and Manipulation**: MongoDB provides operators and methods to interact with arrays, including adding, updating, and removing elements, as well as querying and aggregating data within arrays.

- To create a collection by the name “food” and then insert documents into the food.
- Db.food.insert({_id:1, fruits:[‘banana’, ’apple’, ‘cherry’]})
- Db.food.insert({_id:2, fruits:[‘orange’, ’butterfruit’, ‘mango’]})
- Db.food.insert({_id:3, fruits:[‘pineapple’, ’strawberry’, ‘grapes’]})
- Db.food.insert({_id:4, fruits:[‘banana’, ’straberry, ‘grapes’]})
- Db.food.insert({_id:5, fruits:[‘orange, ’grapes’]})

- Db.food.find({})

1. Find from the food collection which has fruits array constituted of “banana”, “apple”, “cherry”
2. Find from food collection which has fruits array having “banana” as an element
3. Find from food collection which has fruits array having “grapes” in the first index position.
4. Find from the food collection where the size of the array is 2. the size implies that the array holds only two values.

5. To find the document with (_id:1) from the food collection and display the first two elements from the fruits array.

Ans: db.food.find({_id:1},{“fruits”:{$slice:2}})

6. To find all documents from the collection which have elements “orange” and “ grapes” in the array

Ans: db.food.find({fruits:{$all:[“orange”, “grapes”]}}). Pretty()

7. To find documents from collection which have the element “ orange” in the o^{th} index position

Update on the array

- To update the document with _id:1 and replace the element “apple” of the fruits array with “ An apple”

Ans: db.food.update({_id:1, ‘fruits’:’apple’}, {$set:{‘fruits.$:’An apple’}})

- The $ character is a placeholder that refers to the matched element in the array.

Aggregate and MapReduce Function

- Aggregation operations process data records and return computed results.
- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

- Consider the example

```
db.sales.aggregate([{ $group: { _id: "$product",
  totalSales: { $sum: { $multiply: ["$quantity", "$price"] }
  }} }])
```

- This aggregation pipeline will group the data by the "product" field and calculate the total sales for each product by summing the result of multiplying the "quantity" and "price" for each sale.

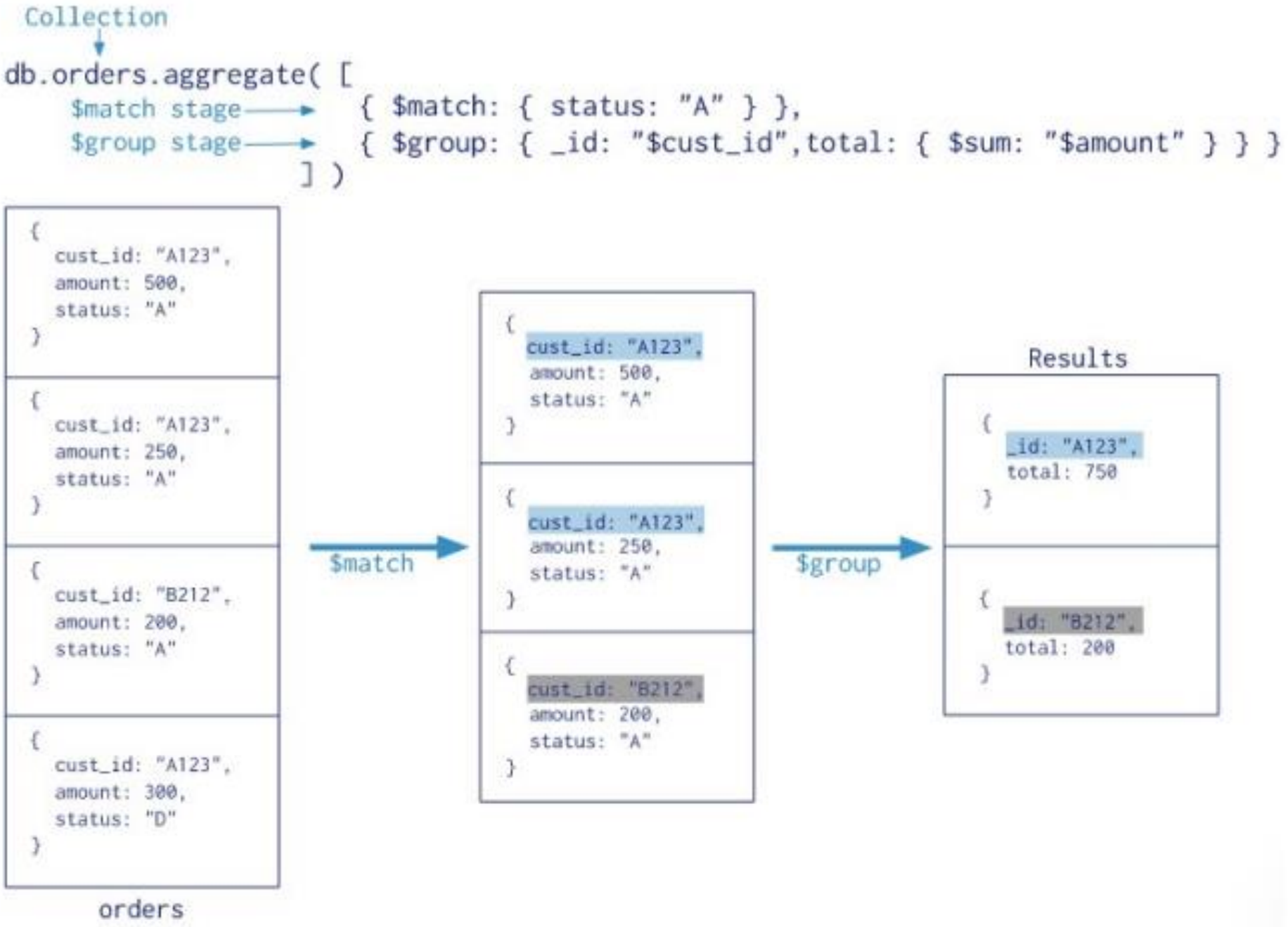
- MongoDB provides three ways to perform aggregation:
- The aggregation pipeline
- The map-reduce function and
- Single purpose aggregation methods.

Aggregation Pipeline

- MongoDB's aggregation framework is modeled on the concept of data processing pipelines.
- Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.
- The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document.
- pipeline operations provide tools for grouping and sorting documents by specific field

```
db.sales.aggregate([
  // Stage 1: Group by product and calculate total revenue
   for each product
   { $group: { _id: "$product", totalRevenue: { $sum: {
   $multiply: ["$quantity", "$price"] } } }},
  // Stage 2: Sort the results by total revenue in
   descending order
  {$sort: { totalRevenue: -1 }}])
```

- Pipeline stages can use operators for tasks such as calculating the average or concatenating a string.
- The aggregation process in MongoDb consists of several stages, each stage transforming the data in some way
- Mongodb provides several built in functions perform various operations such as group, sum, avg, min, max


Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

- Db.teachers.aggragate({$match:{gender:”male”}})
- Db.teachers.aggregate({$group:{_id:”$age”}}

Now group teachers by age and show all teachers names per age group

- Db.techers.aggregate({$group:{_id:”$age”,names:{$push:”$name”}}})
- Here the names field uses the $push operator to add the name field from each document in the group to an array.

Pipeline:

- The MongoDB aggregation pipeline consists of stages.
- Each stage transforms the documents as they pass through the pipeline.
- Pipeline stages do not need to produce one output document for every input document;
- e.g., some stages may generate new documents or filter out documents. Pipeline stages can appear multiple times in the pipeline.
- MongoDB provides the db.collection.aggregrate( ) method in the mongo shell and the aggregate command for aggregation pipeline.

Pipeline Expressions

- The aggregation pipeline in MongoDB consists of multiple stages, and Pipeline Expressions are a key component of these stages.
- You use pipeline expressions to define the transformations and operations that should be applied to the data as it flows through the pipeline.
- These expressions are used to specify the criteria for filtering, reshaping documents, performing mathematical operations, grouping data, and more.

- **$match:** This stage filters documents based on specified criteria, allowing you to select a subset of documents that match certain conditions.
- Suppose you have a collection of "products," and you want to aggregate data for products that have a price greater than $50.

```
db.products.aggregate([ { $match: { price: { $gt: 50 }}},
  // Other aggregation stages can follow here ])
```

- The $match stage filters the documents from the "products" collection, selecting only those where the "price" field is greater than 50 (using the $gt operator).
- After this stage, only documents with a price higher than $50 will be passed to subsequent aggregation stages.

- **$project:** This stage is used to reshape the documents in the pipeline, including selecting or excluding fields, renaming fields, and creating new computed fields.
- db.users.aggregate([{ $project: { _id: 0, // Exclude the default"_id" field userName: "$name", userEmail: "$email" } }])
- The "userName: "$name" expression renames the "name" field to "userName" in the output.
- The "userEmail: "$email" expression renames the "email" field to "userEmail" in the output.

- **$group:** This stage is used to group documents by one or more fields and perform aggregations (e.g., sum, average) on the grouped data.
- **$sort:** This stage sorts the documents based on the values of one or more fields.
- **$unwind:** This stage is used to deconstruct arrays within documents, creating multiple documents for each element in the array. This is useful for further processing of array elements.

Pipeline Operators and Indexes

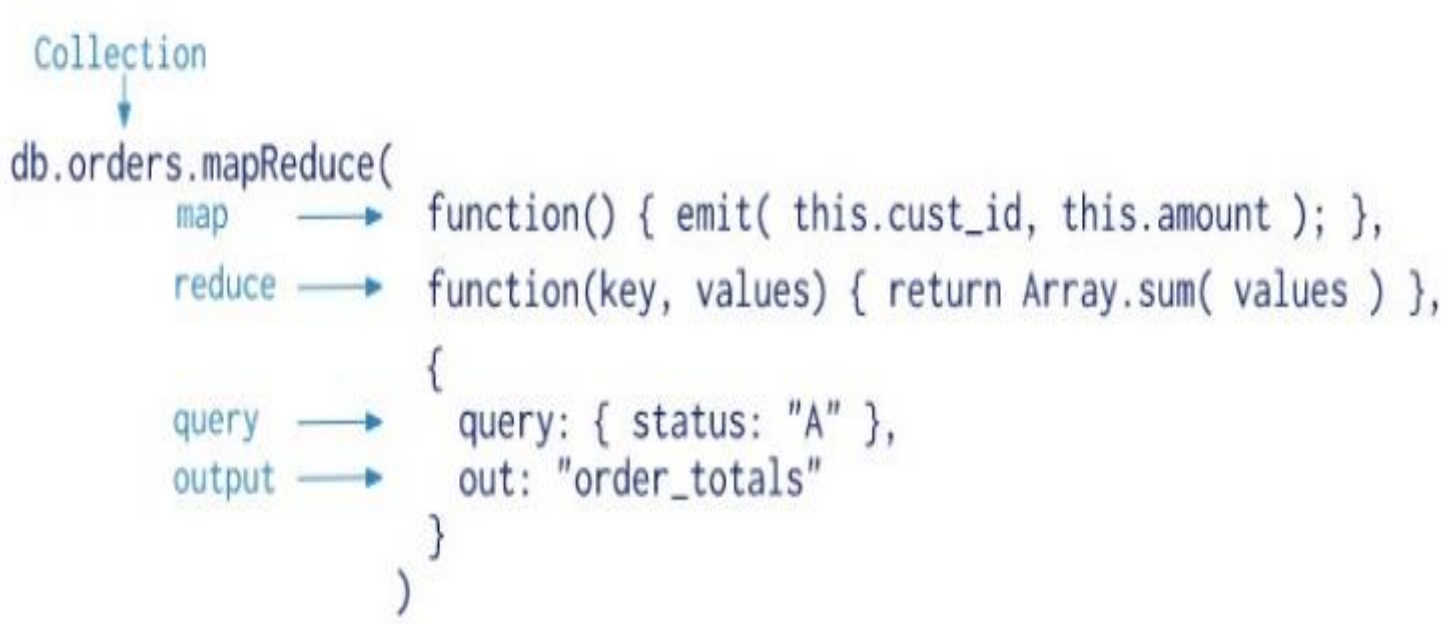
- **Pipeline Operators:**
- Pipeline operators are used in aggregation pipelines to define the various stages of data transformation and processing.
- These operators specify the operations to be performed on the data as it flows through the pipeline. They can include stages like $match, $project, $group, $sort, and many others.

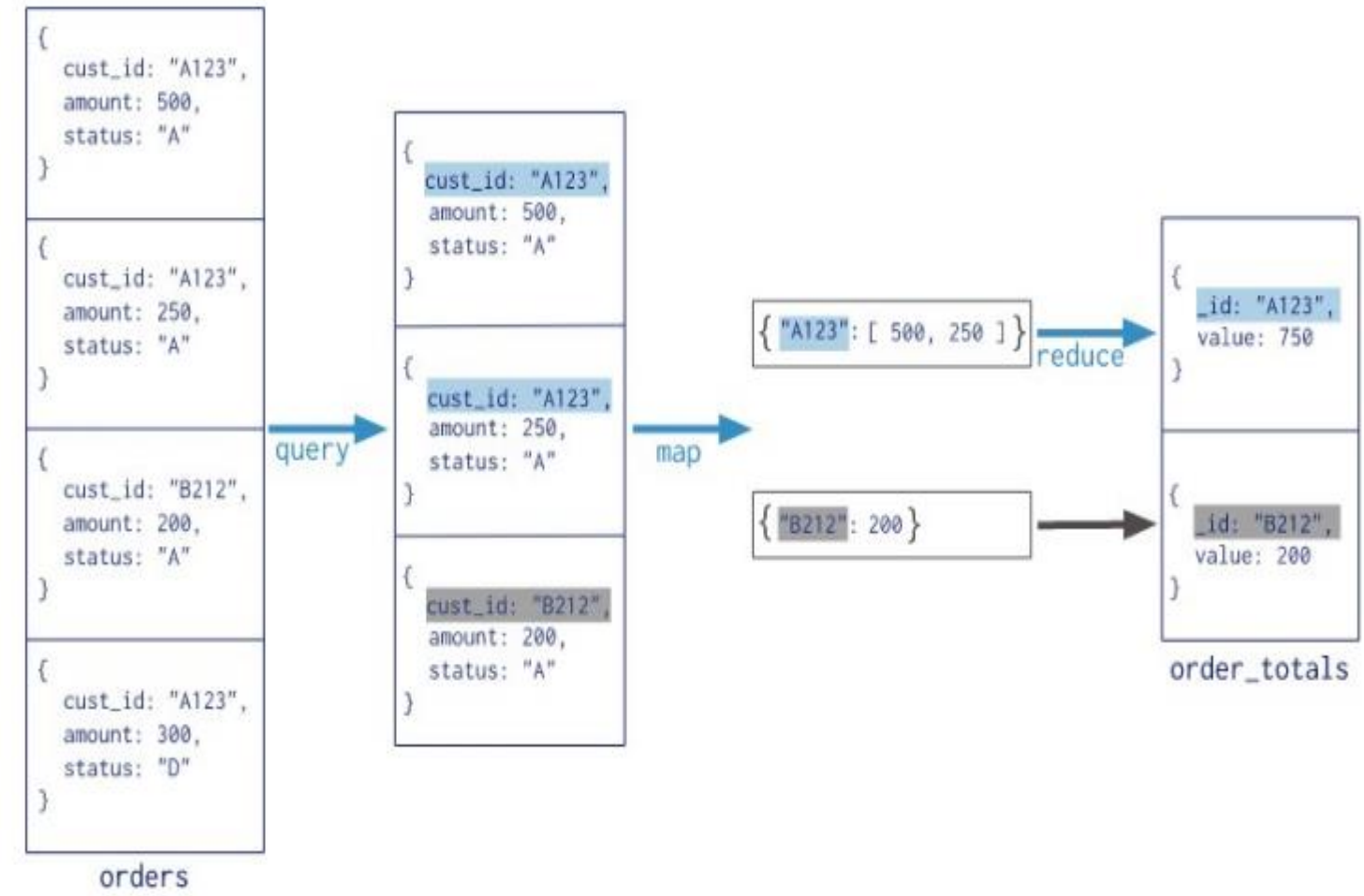
- The $match operator filters documents based on a specified condition.
- It's similar to the find method in regular MongoDB queries.
- db.orders.aggregate([{$match: { status: "open"}}])

- **Indexes:**
- Indexes are database structures that provide a way to access data more efficiently. They work by creating a sorted data structure for one or more fields in a collection, allowing MongoDB to quickly locate and retrieve documents.
- Indexes can significantly improve query performance, especially for queries that filter, sort, or group data based on specific fields.

Map Reduce:

- Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results.
- For map-reduce operations, MongoDB provides the MapReduce database command


Collection
db.orders.mapReduce(
map
function() { emit(this.cust_id, this.amount); },
reduce
function(key, values) { return Array.sum(values) },
{
query
query: { status: "A" },
output
out: "order_totals"
}
)


{
cust_id: "A123",
amount: 500,
status: "A"
}
{
cust_id: "A123",
amount: 250,
status: "A"
}
{
cust_id: "B212",
amount: 200,
status: "A"
}
{
cust_id: "A123",
amount: 300,
status: "D"
}
orders
query
{
cust_id: "A123",
amount: 500,
status: "A"
}
{
cust_id: "A123",
amount: 250,
status: "A"
}
{
cust_id: "B212",
amount: 200,
status: "A"
}
map
{ "A123": [500, 250] }
{ "B212": 200 }
reduce
{
_id: "A123",
value: 750
}
{
_id: "B212",
value: 200
}
order_totals

- MongoDB applies the map phase to each input document (i.e. the documents in the collection that match the query condition).
- The map function emits key-value pairs.
- For those keys that have multiple values, MongoDB applies the reduce phase, which collects and condenses the aggregated data.
- MongoDB then stores the results in a collection.

- All map-reduce functions in MongoDB are JavaScript and run within the mongod process.
- Map-reduce operations take the documents of a single collection as the input and can perform any arbitrary sorting and limiting before beginning the map stage.

- Map Function: The "map" function is a JavaScript function that you define. It takes input documents from a collection and emits key-value pairs as its output. These key-value pairs are then grouped by key.
- Reduce Function: The "reduce" function is another JavaScript function that you define. It takes the key and an array of values emitted by the "map" function and reduces them to a single value, which is the result of aggregation for that key.

- Finalize Function (Optional): The "finalize" function is an optional JavaScript function that can be used to further process the results after the "reduce" function. It can be used to perform additional calculations or transformations on the aggregated data.
- Output Collection: The results of the Map-Reduce operation are stored in a new collection or an existing one, depending on your configuration.

- Map function:

```
Var map=function(){
                emit(this.custid,this.amount);}
```

- Reduce function:

```
Var reduce= function(key,values){return
   Array.sum(values);}
```

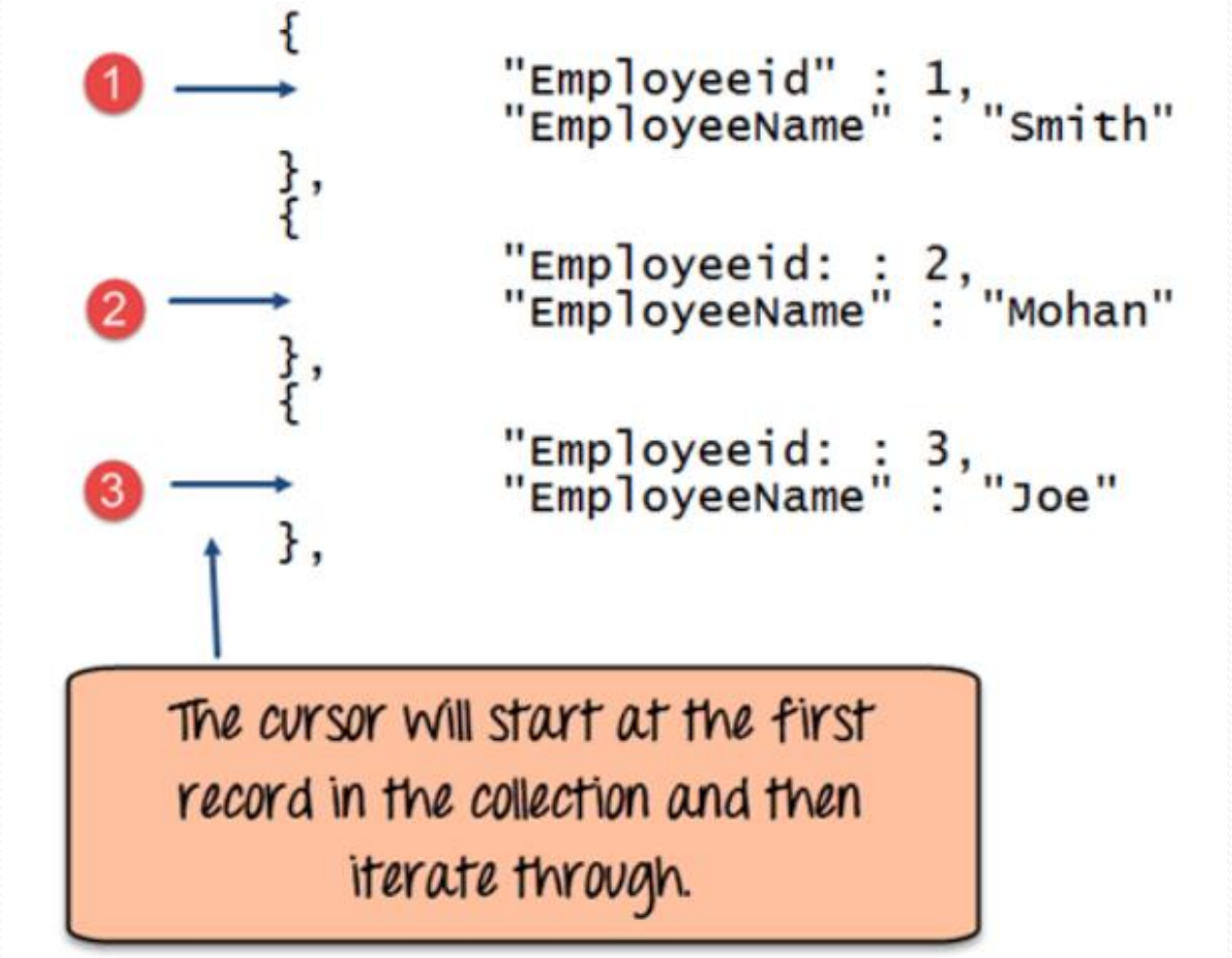
- To execute query:

```
Db.customers.mapReduce(map,reduce.{out:"total",query
   :{acctype:"s"}});
```

MongoDB Cursors

- When the db.collection.find () function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.
- By default, the cursor will be iterated automatically when the result of the query is returned.
- But one can also explicitly go through the items returned in the cursor one by one.

- Query: You send a query to the MongoDB server, specifying your search criteria and any desired options.
- Cursor Creation: The server creates a cursor and returns it to your application. The cursor contains information about the query, such as the criteria and options, but it doesn't contain the actual data.
- Iteration: You can iterate through the cursor to retrieve documents one by one or in batches. This is often done using methods like next() to get the next document in the result set.

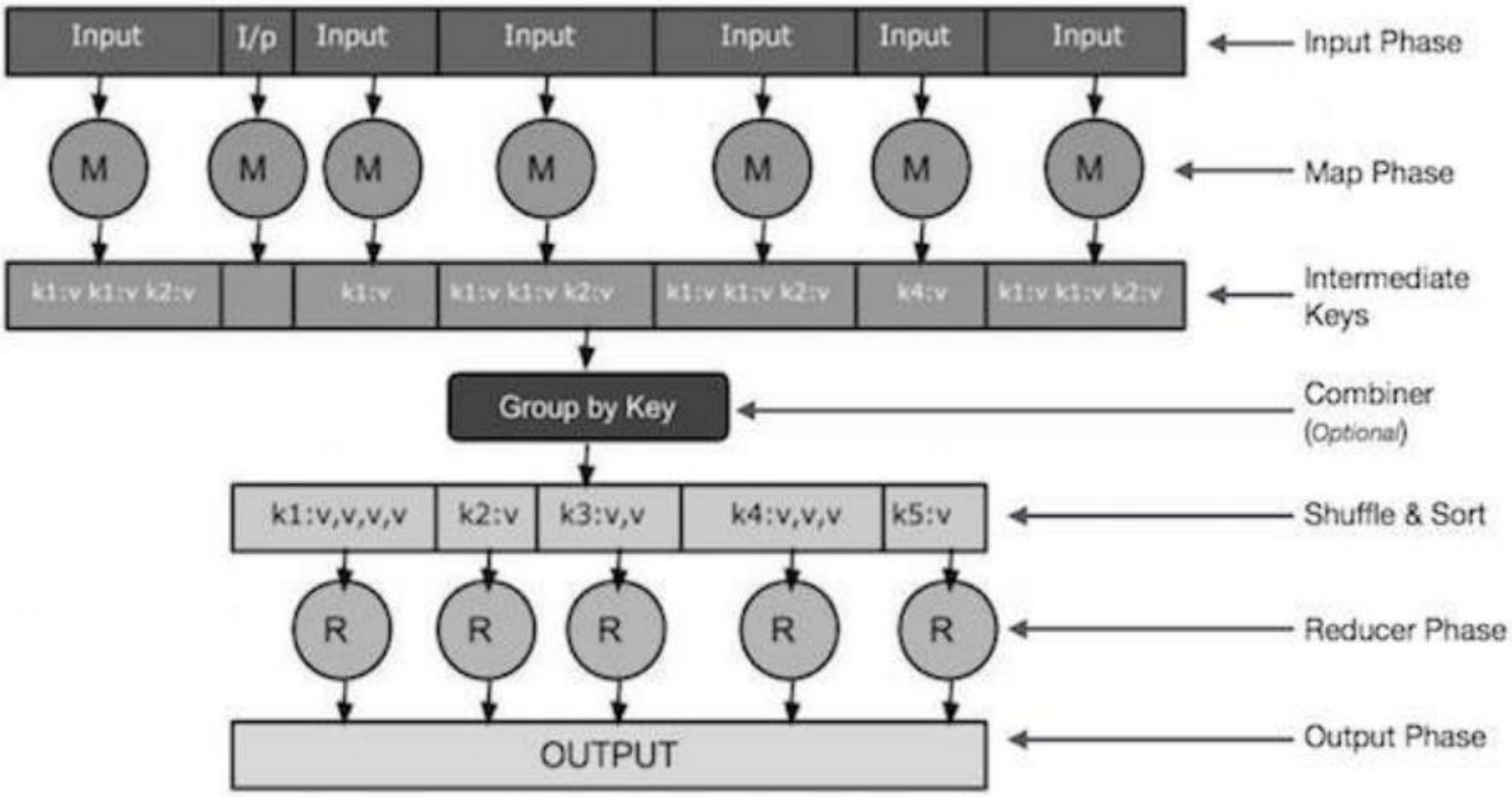

{
1
"Employeeid" : 1,
"EmployeeName" : "Smith"
},
{
"Employeeid: : 2,
2
"EmployeeName" : "Mohan"
},
{
"Employeeid: : 3,
3
"EmployeeName" : "Joe"
},
The cursor will start at the first record in the collection and then iterate through.

- The following example shows how this can be done.

```
var myEmployee = db.Employee.find( { Employeeid :
{ $gt:2 }});
while(myEmployee.hasNext())
 { print(tojson(myEmployee.next())); }
```

Architecture of MapReduce

- The MapReduce algorithm contains two important tasks, namely Map and Reduce.
- The Map task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key-value pairs).
- The Reduce task takes the output from the Map as an input and combines those data tuples (key-value pairs) into a smaller set of tuples.


Input
I/p
Input
Input
Input
Input
Input
Input Phase
M
Map Phase
k1:v k1:v k2:v
k1:v
k1:v k1:v k2:v
k1:v k1:v k2:v
k4:v
k1:v k1:v k2:v
Intermediate Keys
Group by Key
Combiner (Optional)
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
Shuffle & Sort
R
Reducer Phase
OUTPUT
Output Phase

The various phases are

- Input Phase – Here we have a Record Reader that translates each record in an input file and sends the parsed data to the mapper in the form of key-value pairs.
- Map – Map is a user-defined function, which takes a series of key-value pairs and processes each one of them to generate zero or more key-value pairs.
- Intermediate Keys – They key-value pairs generated by the mapper are known as intermediate keys.

- Combiner – A combiner is a type of local Reducer that groups similar data from the map phase into identifiable sets. It takes the intermediate keys from the mapper as input and applies a user-defined code to aggregate the values in a small scope of one mapper.

- Reducer – The Reducer takes the grouped key-value paired data as input and runs a Reducer function on each one of them. Here, the data can be aggregated, filtered, and combined in a number of ways, and it requires a wide range of processing. Once the execution is over, it gives zero or more key-value pairs to the final step.

- Output Phase – In the output phase, we have an output formatter that translates the final key-value pairs from the Reducer function and writes them onto a file using a record writer