**ASSIGNMENT – 4**

**Aim:** Implement Map-reduce and aggregation, indexing with suitable example in MongoDB. Demonstrate the following:

* Aggregation framework
* Create and drop different types of indexes and explain () to show the advantage of the indexes.

**Theory:**

**Map-Reduce**

MongoDB also provides map-reduce operations to perform aggregation. In general, map-reduce operations have two phases: a map stage that processes each document and emits one or more objects for each input document, and reduce phase that combines the output of the map operation. Optionally, map-reduce can have a finalize stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional finalize operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, MapReduce is less efficient and more complex than the aggregation pipeline.

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](https://docs.mongodb.com/manual/reference/command/mapReduce/#dbcmd.mapReduce) database command.

**Aggregation Pipeline as Alternative to Map-Reduce**

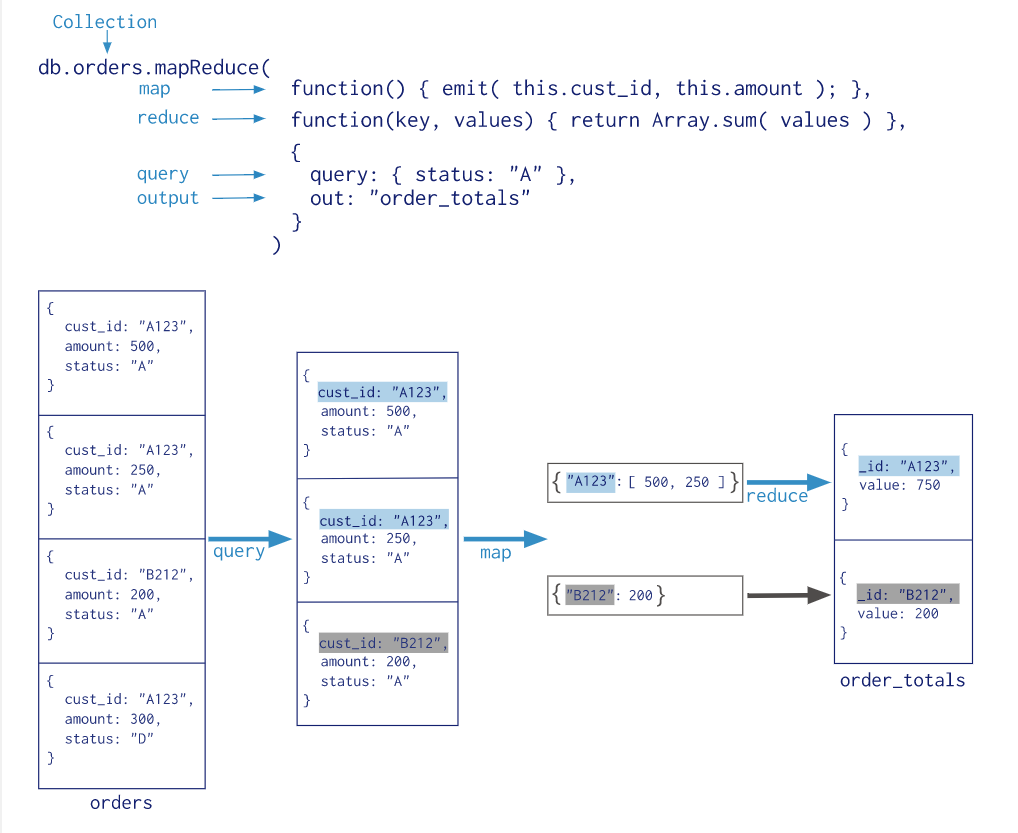
An aggregation pipeline provides better performance and usability than a map-reduce operation.

Map-reduce operations can be rewritten using aggregation pipeline operators, such as $group, $merge, and others.

For map-reduce operations that require custom functionality, MongoDB provides the $accumulator and $function aggregation operators starting in version 4.4. Use these operators to define custom aggregation expressions in JavaScript.

In mongosh, the db.collection.mapReduce() method is a wrapper around the MapReduce command. The following examples use the db.collection.mapReduce() method.

Consider the following map-reduce operation:



## **Map-Reduce Behavior**

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results.

When returning the results of a map reduce operation inline, the result documents must be within the [BSON Document Size](https://docs.mongodb.com/manual/reference/limits/#BSON-Document-Size) limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations.

MongoDB supports map-reduce operations on [sharded collections](https://docs.mongodb.com/manual/sharding/). Map-reduce operations can also output the results to a sharded collection.

# **Map Reduce Concurrency**

The map-reduce operation is composed of many tasks, including reads from the input collection, executions of the map function, executions of the reduce function, writes to a temporary collection during processing, and writes to the output collection.

During the operation, map-reduce takes the following locks:

* The read phase takes a read lock. It yields every 100 documents.
* The insert into the temporary collection takes a write lock for a single write.
* If the output collection does not exist, the creation of the output collection takes a write lock.
* If the output collection exists, then the output actions (i.e. merge, replace, reduce) take a write lock. This write lock is global, and blocks all operations on the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/#bin.mongod) instance

Create a sample collection orders with these documents:

|  |
| --- |
| db.orders.insertMany([ |
| { \_id: 1, cust\_id: **"Ant O. Knee"**, ord\_date: new Date(**"2020-03-01"**), price: 25, items: [ { sku: **"oranges"**, qty: 5, price: 2.5 },  { sku: **"apples"**, qty: 5, price: 2.5 } ], status: **"A"** }, |
| { \_id: 2, cust\_id: **"Ant O. Knee"**, ord\_date: new Date(**"2020-03-08"**), price: 70, items: [ { sku: **"oranges"**, qty: 8, price: 2.5 },  { sku: **"chocolates"**, qty: 5, price: 10 } ], status: **"A"** }, |
| { \_id: 3, cust\_id: **"Busby Bee"**, ord\_date: new Date(**"2020-03-08"**), price: 50, items:  [ { sku: **"oranges"**, qty: 10, price: 2.5 },  { sku: **"pears"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 4, cust\_id: **"Busby Bee"**, ord\_date: new Date(**"2020-03-18"**), price: 25, items:  [ { sku: **"oranges"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 5, cust\_id: **"Busby Bee"**, ord\_date: new Date(**"2020-03-19"**), price: 50, items:  [ { sku: **"chocolates"**, qty: 5, price: 10 } ], status: **"A"**}, |
| { \_id: 6, cust\_id: **"Cam Elot"**, ord\_date: new Date(**"2020-03-19"**), price: 35, items:  [ { sku: **"carrots"**, qty: 10, price: 1.0 },  { sku: **"apples"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 7, cust\_id: **"Cam Elot"**, ord\_date: new Date(**"2020-03-20"**), price: 25, items:  [ { sku: **"oranges"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 8, cust\_id: **"Don Quis"**, ord\_date: new Date(**"2020-03-20"**), price: 75,  items: [ { sku: **"chocolates"**, qty: 5, price: 10 }, { sku: **"apples"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 9, cust\_id: **"Don Quis"**, ord\_date: new Date(**"2020-03-20"**), price: 55,  items: [ { sku: **"carrots"**, qty: 5, price: 1.0 }, { sku: **"apples"**, qty: 10, price: 2.5 }, { sku: **"oranges"**, qty: 10, price: 2.5 } ], status: **"A"** }, |
| { \_id: 10, cust\_id: **"Don Quis"**, ord\_date: new Date(**"2020-03-23"**), price: 25, items:  [ { sku: **"oranges"**, qty: 10, price: 2.5 } ], status: **"A"** } |
| ]) |

**Display the Total Price Per Customer**

Perform the map-reduce operation on the orders collection to group by the cust\_id.

1. Define the map function to process each input document:
   * In the function, this refers to the document that the map-reduce operation is processing.
   * The function maps the items to the cust\_id for each document and emits the cust\_id and items.

|  |
| --- |
| **var mapFunction1 = function() {** |
| **emit(this.cust\_id, this.items);** |
| **};** |

1. Define the corresponding reduce function with two arguments keyCustId and valuesitems:
   * The valuesitems is an array whose elements are the items values emitted by the map function and grouped by keyCustId.
   * The function reduces the valuesitems array to the sum of its elements.

|  |
| --- |
| **var reduceFunction1 = function(keyCustId, valuesPrices) {** |
| **return keyCustId;** |
| **};** |

1. Perform map-reduce on all documents in the orders collection using the map map function and the reduce  reduce function:

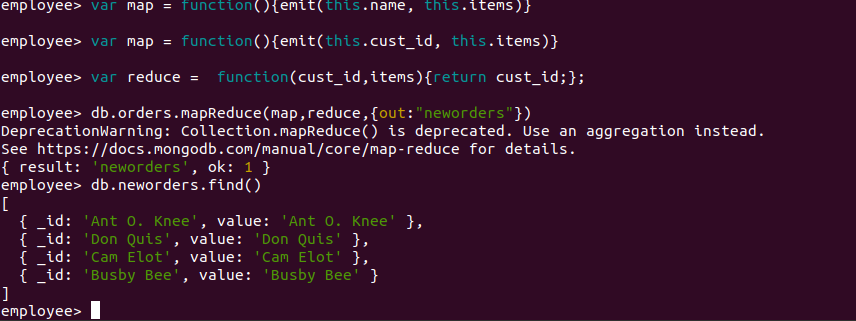
|  |
| --- |
| **db.orders.mapReduce(** |
| **map,** |
| **reduce,** |
| **{ out: "new\_orders " }** |
| **)** |

1. This operation outputs the results to a collection named **new\_orders**. If the **new\_orders** collection already exists, the operation will replace the contents with the results of this map-reduce operation.
2. Query the **new\_orders** collection to verify the results:

|  |
| --- |
| **db. new\_orders.find().sort( { \_id: 1 } )** |

1. The operation returns these documents:

|  |
| --- |
|  |
|  |
|  |
|  |



**Aggregation Framework**

**Aggregation**

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

* Group values from multiple documents together.
* Perform operations on the grouped data to return a single result.
* Analyze data changes over time.

### Aggregation Pipeline

The following aggregation pipeline contains two stages and returns the total quantity of urgent orders for each product:

|  |
| --- |
| db.orders.aggregate( [ |
| { $match: { status: **"urgent"** } }, |
| { $group: { \_id: **"$productName"**, sumQuantity: { $sum: **"$quantity"** } } } |
| ] ) |

The $match stage:

* Filters the documents to those with a status of urgent.
* Outputs the filtered documents to the $group stage.

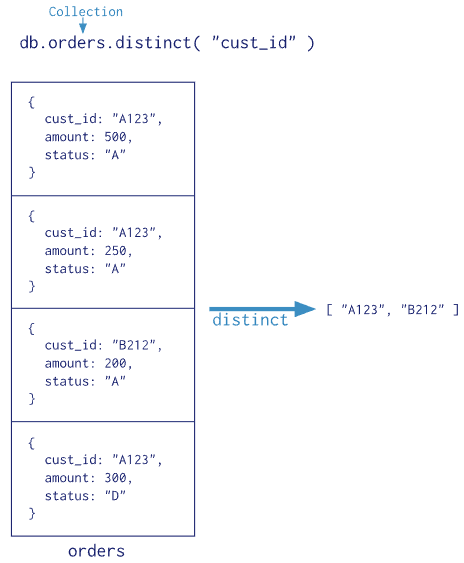
The $group stage:

* Groups the input documents by productName.
* Uses $sum to calculate the total quantity for each productName, which is stored in the sumQuantity field returned by the aggregation pipeline.

## **Single Purpose Aggregation Operations**

MongoDB provides also db.collection.estimatedDocumentCount(), db.collection.count() and db.collection.distinct().

All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of an aggregation pipeline.



Using the available aggregation pipeline operators, you can rewrite the map-reduce operation without defining custom functions:

|  |
| --- |
| **db.orders.aggregate([** |
| **{ $group: { \_id: "$cust\_id", value: { $sum: "$price" } } },** |
| **{ $out: "agg\_frame\_1" }** |
| **])** |

1. The $group stage groups by the cust\_id and calculates the value field (See also $sum). The value field contains the total price for each cust\_id.

The stage output the following documents to the next stage:

|  |
| --- |
| { **"\_id"** : **"Don Quis"**, **"value"** : 155 } |
| { **"\_id"** : **"Ant O. Knee"**, **"value"** : 95 } |
| { **"\_id"** : **"Cam Elot"**, **"value"** : 60 } |
| { **"\_id"** : **"Busby Bee"**, **"value"** : 125 } |

1. Then, the $out writes the output to the collection agg\_alternative\_1. Alternatively, you could use $merge instead of $out.
2. Query the agg\_alternative\_1 collection to verify the results:

|  |
| --- |
| **db.agg\_frame\_1.find().sort( { \_id: 1 } )** |

1. The operation returns the following documents:

|  |
| --- |
| { **"\_id"** : **"Ant O. Knee"**, **"value"** : 95 } |
| { **"\_id"** : **"Busby Bee"**, **"value"** : 125 } |
| { **"\_id"** : **"Cam Elot"**, **"value"** : 60 } |
| { **"\_id"** : **"Don Quis"**, **"value"** : 155 }  db.orders.aggregate( [  // Stage 1: Filter pizza order documents by date range  {  $match:  {  "date": { $gte: new ISODate( "2020-01-30" ), $lt: new ISODate( "2022-01-30" ) }  }  },  // Stage 2: Group remaining documents by date and calculate results  {  $group:  {  \_id: { $dateToString: { format: "%Y-%m-%d", date: "$date" } },  totalOrderValue: { $sum: { $multiply: [ "$price", "$quantity" ] } },  averageOrderQuantity: { $avg: "$quantity" }  }  },  // Stage 3: Sort documents by totalOrderValue in descending order  {  $sort: { totalOrderValue: -1 }  }  ] )   |  | | --- | | [ | | { \_id: **'2022-01-12'**, totalOrderValue: 790, averageOrderQuantity: 30 }, | | { \_id: **'2021-03-13'**, totalOrderValue: 770, averageOrderQuantity: 15 }, | | { \_id: **'2021-03-17'**, totalOrderValue: 630, averageOrderQuantity: 30 }, | | { \_id: **'2021-01-13'**, totalOrderValue: 350, averageOrderQuantity: 10 } | | ] | |

**Indexes:**

Indexes provide high performance read operations for frequently used queries. This section introduces indexes in mongodb, describes the types and configuration options for indexes, and describes special types of indexing mongodb supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes. Indexes support the efficient execution of queries in mongodb.Without indexes, mongodb must scan every document in a collection to select those documents that match the query statement. These collection scans are inefficient because they require mongod to process a larger volume of data than an index for each operation. Indexes are special data structures that store a small portion of the collection’s data set in an easy to traverse form.

The index stores the value of a specific field or set of fields, ordered by the value of the field.

**Create an Index**

To create an index in the Mongo Shell, use db.collection.createIndex().

**db.collection.createIndex( <key and index type specification>, <options> )**

The following example creates a single key descending index on the name field:

**db.collection.createIndex( { name: -1 } )**

### Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index ( i.e. 1 or -1) using underscores as a separator. For example, an index created on { item : 1, quantity: -1 } has the name item\_1\_quantity\_-1.

You can create indexes with a custom name, such as one that is more human-readable than the default. For example, consider an application that frequently queries the products collection to populate data on existing inventory. The following createIndex() method creates an index on item and quantity named query for inventory:

|  |
| --- |
| db.products.createIndex( |
| { item: 1, quantity: -1 } , |
| { name: **"query for inventory"** } |
| ) |

**Index Types**

Mongodb provides a number of different index types to support specific types of data and queries.

Default \_id

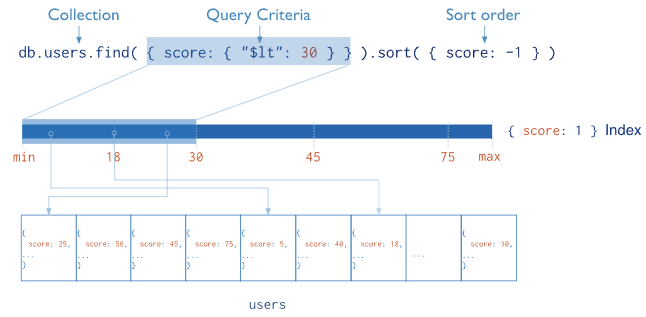
All mongodb collections have an index on the \_id field that exists by default. If applications do not specify a value for \_id the driver or the mongod will create an \_id field with an objectid value.

The \_id index is unique, and prevents clients from inserting two documents with the same value for the \_id field.

**Compound Index**

Mongodb also supports user-defined indexes on multiple fields. These compound indexes behave like single-field indexes; however, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of {userid: 1, score:-1}, the index sorts first by userid and then, within each userid value, sort by score.

Consider the following Illustration of this compound index:



**Multi key Index**

Mongodb uses multikey indexes to index the content stored in arrays. If you index a field that holds an array value, mongodb creates separate index entries for every element of the array. These multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays. Mongodb automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

**Conclusion:** Thus we implemented map reduce, Aggregation framework and indexing using MongoDB.