Assignment No. 2

Problem statement:Aggregation with suitable example using MongoDB.

Objectives: To understand the basicAggregation queries of MongoDB.

Theory:
Aggregation Pipeline
Aggregation pipeline gives you a way to transform and combine documents in your collection. You do it by passing the documents through a pipeline that's somewhat analogous to the Unix "pipe" where you send output from one command to another to a third, etc.
The simplest aggregation you are probably already familiar with is the SQL group by expression. We already saw the simple count() method, but what if we want to see how many student are male and how many are female?

```
db.student.aggregate ({$group :{_id:'$gender',total: {$sum:1}}})
```

In the shell we have the aggregate helper which takes an array of pipeline operators. For a simple count grouped by something, we only need one such operator and it's called $group. This is the exact analog of GROUP BY in SQL where we create a new document with _id field indicating what field we are grouping by (here it's gender) and other fields usually getting assigned results of some aggregation, in this case we $sum 1 for each document that matches a
particular gender.
You probably noticed that the _id field was assigned '$gender' and not 'gender' - the '$' before a field name indicates that the value of this field from incoming document will be substituted. What are some of the other pipeline operators that we can use? The most common one to use before (and frequently after) $group would be $match - this is exactly like the find method and it allows us to aggregate only a matching subset of our documents, or to exclude some documents from our result.

```
db.student.aggregate ({$match: {weight :{$lt:600}}},
                      {$group: {_id:'$gender', total :{$sum:1},
avgVamp :{$avg:'$vampires'}}}, {$sort:{avgVamp:-1}} )
```

Here we introduced another pipeline operator $sort which does exactly what you would expect, along with it we alsoget $skip and $limit. We also used a $group operator $avg.
MongoDB arrays are powerful and they don't stop us from being able to aggregate on values that are stored inside ofthem. We do need to be able to "flatten" them to properly count everything:

```
db.student.aggregate ({$unwind: 'like'},
{$group: {_id:'$like, total :{$sum:1},student:{$addToSet:'$like'}}},
{$sort :{ total:-1}},{$limit:1} )
```

Here we will find out which food item is loved by the most student and we will also get the list of names of all thestudent that like it. $sort and $limit in combination allow you to get answers to "top N" types of questions.

$sort:-

The $sort stage has the following prototype form:
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
$sort takes a document that specifies the field(s) to sort by and the respective sort order. <sort order> can have one of the following values:
1 to specify ascending order.
-1 to specify descending order.
{ $meta: "textScore" } to sort by the computed textScore metadata in descending order.
To ascending order for a field or fields to sort by and a value of 1 or -1 to specify an ascending or descending sort respectively, as in the following example:

```
db.users.aggregate (
[
{$sort: {age: -1, posts: 1}}
]
)
```

$limit:-
Limits the number of documents passed to the next stage in the *pipeline*.
The $limit stage has the following prototype form:
{$limit:<positiveinteger>}
$limit takes a positive integer that specifies the maximum number of documents to pass along.
Example
Consider the following example:

```
db.article.aggregate(
{$limit:5}
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit has no effect on the content of the documents it passes.

There is another powerful pipeline operator called $project (analogous to the projection we can specify to find) whichallows you not just to include certain fields, but to create or calculate new fields based on values in existing fields. For23
example, you can use math operators to add together values of several fields before finding out the average, or youcan use string operators to create a new field that's a concatenation of some
existing fields.

$project:-
The $project takes a document that can specify the inclusion of fields, the suppression of the _id field, the addition of new fields, and the resetting the values of existing fields. The specifications have the following forms:

Syntax	Description
<field>: <1 or true>	Specify the inclusion of a field.
_id: <0 or false>	Specify the suppression of the _id field.
<field>: <expression>	Add a new field or reset the value of an existing field.

The following $project stage includes only the _id, title, and the author fields in its output documents:

```
db.books.aggregate( [ { $project : { title : 1 , author : 1 } } ] )
```

To include only the title field in the embedded document in the stop field, you can use the dot notation:
db.bookmarks.aggregate([{ $project: { "stop.title": 1 } }])
Or, you can nest the inclusion specification in a document:
db.bookmarks.aggregate([{ $project: { stop: { title: 1 } } }])

$group :-
Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an_id field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group‘s _id field. $group does *not* order its output documents.
The $group stage has the following prototype form:
{$group:{_id:<expression>,<field1>:{<accumulator1>:<expression1>},...}}
The _id field is *mandatory*; however, you can specify an_id value of null to calculate accumulated values for all the input documents as a whole.
The remaining computed fields are *optional* and computed using the <accumulator> operators.
The _id and the <accumulator> expressions can accept any valid *expression*. For more information on expressions, see *Expressions*.
Accumulator Operator
The <accumulator> operator must be one of the following accumulator operators:

Name	Description
$addToSet	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.
$avg	Returns an average for each group. Ignores non-numeric values.
$first	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
$last	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max	Returns the highest expression value for each group.
$min	Returns the lowest expression value for each group.
$push	Returns an array of expression values for each group.
$sum	Returns a sum for each group. Ignores non-numeric values.

$group Operator and Memory
The $group stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, $group will produce an error. However, to allow for the handling of large datasets, set the allowDiskUse option to true to enable $group operations to write to temporary files. See db.collection.aggregate() method and the aggregate command for details.
Examples
Calculate Count, Sum, and Average
Given a collection sales with the following documents:
{"_id":1,"item":"abc","price":10,"quantity":2,"date":ISODate("2014-03-1T08:00:00Z")}
{"_id":2,"item":"jkl","price":20,"quantity":1,"date":ISODate("2014-03-01T09:00:00Z")}
{"_id":3,"item":"xyz","price":5,"quantity":10,"date":ISODate("2014-03-15T09:00:00Z")}
{"_id":4,"item":"xyz","price":5,"quantity":20,"date":ISODate("2014-04-04T11:21:39.736Z")}

{"_id":5,"item":"abc","price":10,"quantity":10,"date":ISODate("2014-04-04T21:23:13.331Z")}

Group by Month, Day, and Year

The following aggregation operation uses the $group stage to group the documents by the month, day, and year and calculates the total price and the average quantity as well as counts the documents per each group:

```
db.sales.aggregate(
[
{
$group:{
_id:{month:{$month:"$date"},day:{$dayOfMonth:"$date"},year:{$year:"$date"}},
totalPrice:{$sum:{$multiply:["$price","$quantity"]}},
averageQuantity:{$avg:"$quantity"},
count:{$sum:1}
}
}
]
)
```

The operation returns the following results:

```
{"_id":{"month":3,"day":15,"year":2014},"totalPrice":50,"averageQuantity":10,"count":1}
{"_id":{"month":4,"day":4,"year":2014},"totalPrice":200,"averageQuantity":15,"count":2}
{"_id":{"month":3,"day":1,"year":2014},"totalPrice":40,"averageQuantity":1.5,"count":2}
```

Group by null

The following aggregation operation specifies a group _id of null, calculating the total price and the average quantity as well as counts for all documents in the collection:

```
db.sales.aggregate(
[
{
$group:{
_id:null,
totalPrice:{$sum:{$multiply:["$price","$quantity"]}},
averageQuantity:{$avg:"$quantity"},
count:{$sum:1}
}
}
]
)
```

The operation returns the following result:

```
{"_id":null,"totalPrice":290,"averageQuantity":8.6,"count":5}
```

Retrieve Distinct Values

Given a collection sales with the following documents:

```
{"_id":1,"item":"abc","price":10,"quantity":2,"date":ISODate("2014-03-01T08:00:00Z")}
{"_id":2,"item":"jkl","price":20,"quantity":1,"date":ISODate("2014-03-01T09:00:00Z")}
{"_id":3,"item":"xyz","price":5,"quantity":10,"date":ISODate("2014-03-15T09:00:00Z")}
{"_id":4,"item":"xyz","price":5,"quantity":20,"date":ISODate("2014-04-04T11:21:39.736Z")}
{"_id":5,"item":"abc","price":10,"quantity":10,"date":ISODate("2014-04-04T21:23:13.331Z")}
```

The following aggregation operation uses the $group stage to group the documents by the item to retrieve the distinct item values:

```
db.sales.aggregate([{$group:{_id:"$item"}}])
```

The operation returns the following result:

{"_id":"xyz"}
{"_id":"jkl"}
{"_id":"abc"}

$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.
The $unwind stage has the following prototype form:
{ $unwind: <field path> }
To specify a field path, prefix the field name with a dollar sign $ and enclose in quotes.
If a value in the field specified by the field path is *not* an array, db.collection.aggregate() generates an error.
If you specify a path for a field that does not exist in an input document, the pipeline ignores the input document and will not output documents for that input document.
If the array holds an empty array ([]) in an input document, the pipeline ignores the input document and will not output documents for that input document.
Examples
Consider an inventory with the following document:
{"_id":1,"item":"ABC1",sizes:["S","M","L"]}
The following aggregation uses the $unwind stage to output a document for each element in the sizes array:
db.inventory.aggregate([{$unwind:"$sizes"}])
The operation returns the following results:
{"_id":1,"item":"ABC1","sizes":"S"}
{"_id":1,"item":"ABC1","sizes":"M"}
{"_id":1,"item":"ABC1","sizes":"L"}
Each document is identical to the input document except for the value of the sizes field which now holds a value from the original sizes array.
$skip:-
Skips over the specified number of *documents* that pass into the stage and passes the remaining documents to the next stage in the *pipeline*.
The $skip stage has the following prototype form:
{$skip:<positiveinteger>}
$skip takes a positive integer that specifies the maximum number of documents to skip.
Example
Consider the following example:
db.article.aggregate({$skip:5});
This operation skips the first 5 documents passed to it by the pipeline. $skip has no effect on the content of the documents it passes along the pipeline.
$limit:-
Limits the number of documents passed to the next stage in the*pipeline*.
The $limit stage has the following prototype form:
{$limit:<positiveinteger>}
$limit takes a positive integer that specifies the maximum number of documents to pass along.
Example
Consider the following example:
db.article.aggregate(
{$limit:5}
);

This operation returns only the first 5 documents passed to it from by the pipeline. $limit has no effect on the content of the documents it passes.

Conclusion:

Assignment No.

Problem statement: Map-reduce with suitable example using MongoDB.

Objectives: To understand the basic map-reduce of MongoDB

MapReduce

MapReduce is the Uzi of aggregation tools. Everything described with count, distinct, and group can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem, sends chunks of it to different machines, and lets each machine solve its part of the problem. When all of the machines are finished, they merge all of the pieces of the solution back into a full solution.

MapReduce has a couple of steps. It starts with the map step, which maps an operation onto every document in a collection. That operation could be either "do nothing" or "emit these keys with X values." There is then an intermediary stage called the shuffle step: keys are grouped and lists of emitted values are created for each key. The reduce takes this list of values and reduces it to a single element. This element is returned to the shuffle step until each key has a list containing a single value: the result. The price of using MapReduce is speed: group is not particularly speedy, but MapReduce is slower and is not supposed to be used in "real time." You run
Map-reduceis a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses mapReduce command for map-reduce operations. MapReduce is generally used for processing large data sets.
MapReduce Command:
Following is the syntax of the basic mapReduce command:

```
>db.collection.mapReduce(
function(){emit(key,value);},//map function
function(key,values){returnreduceFunction},//reduce function
{
out: collection,
query: document,
sort: document,
limit: number
}
)
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.
In the above syntax:
map is a javascript function that maps a value with a key and emits a key-valur pair
reduce is a javscript function that reduces or groups all the documents having the same key
out specifies the location of the map-reduce query result
query specifies the optional selection criteria for selecting documents
sort specifies the optional sort criteria
limit specifies the optional maximum number of documents to be returned
Emit takes a key (used to reduce the data) and a value.

It's important to note that MongoDB only calls the reduce function for those keys that have multiple values. Therefore the value you emit should mirror the return value from the reduce function.
we will use a mapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code:

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
    {
query:{status:"active"},
out:"post_total"
    }
)
```

The above mapReduce query outputs the following result:

```
{
  "result" : "post_total",
  "timeMillis" : 9,
  "counts" : {
    "input" : 4,
    "emit" : 4,
    "reduce" : 2,
    "output" : 2
  },
  "ok" : 1,

}
```

The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.
To see the result of this mapReduce query use the find operator:

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
    {
query:{status:"active"},
out:"post_total"
    }
).find()
```

The above query gives the following result which indicates that both userstom and mark have two posts in active states:

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

Conclusion: