**Performance:**

**1)Introduction:**

## This week we understand aggregation framework of mongo. The aggregation framework has its roots in SQL group by clause. We will use this framework to calculate sums, counts, avg and other functions where grouped by certain value.

**2) Simple Aggregation Example :**

## 

## Lets start with a simple example of aggregation framework. Lets start by consider a SQL table project as shown below. In order to fetch count of the products owns by each manufacture separately, then we have to use group by clause like the below query.

## Macintosh HD:Users:455814:Desktop:Screen Shot 2016-02-06 at 1.26.29 am.png

## The same can be done in mongoDb like:

## 

## db.products.aggregate([

## { $group :

## {

## \_id : “$manufacturer”

## num\_products:{$sum : 1}

## }

## 

## }

## ])

## Aggregate(): aggregate is a function in MongoDb to use aggregrate Operation. This function takes array of documents as parameters.

## $group: this operator is used when we wanted to do a group by operation.

## \_id: specifies with respective which column we are trying to do group by.

## $sum : this parameter is to sum the value. In this place, we are adding with 1 every time and saving the data into num\_products.

**3) The Aggregration Framework:**

## Aggregation Framework used a pipeline in MongoDB. These are similar to the pipes that we use in Unix. The pipes are used to divide and task into stages and then moving the input from one stage to other by manipulating the data.

## At the top is a collection, which moved to a series of stages and then gives the result set. Let us go thru the each of these pipeline stages briefly.

## $project: this will reshape the document. It will select certain fields of the documents and also elements deep in the hierarchy and bring it up to the top. One document that enters the projection state will give one document as ouput.

## $match : this is filtering state where we will select the documents that we want to analyze in the aggregation state. This will be n:1 mappin i.e for every n documents as input, we get 1 document as filter.

## $group : this is the central for aggregation. This will help us to do aggregate. Inside the group stage, we can use thing like sum, avg and count that will group to gether the documents and perform aggregation function. By default this is a n:1.

## $sort: this is used to sort the documents : this is 1:1.

## $skip : this is used to skip forward. This is n:1.

## $limit : n:1 .

## $unwind: while working with aggregations and when wanted to group on something of a array then we have to flatten the documents and we use $unwind. This will essentially normalize the data. This is 1:n..

## $out : Usually, mongodb returns the documents to the cursor but if we want to redirect to some that collection then we have use $out. This is 1:.

## Find MoreABout : $redact, $geoNear.

## These stages can appear multiple times inside the pipeline and can be any different order. In aggregation function, we pass array of documents and each of the element in that arrays is a stage in pipeline that gonna transform that collections.

## 

**4) Simple Example Expanded.**

## Lets go thru the first example that we have gone thru in the 2nd chapter. So if we recall, we have started with a products collections and then we did some grouping on the collection, to get some results.

## db.products.aggregate([

## { $group :

## {

## \_id : “$manufacturer”,

## num\_products:{$sum : 1}

## }

## 

## }

## ])

## Explaination : This is the query we used. In the above query we used the aggregate function. We put the collection thru $group stage. And we have only one stage in the aggregate pipeline of the above query.

## We asked it to group by manufacturer and we do that by specifying the *id key. Then we are creating a new key called num\_products, with the operator $sum.*

## So what logically happens here?

## So the aggregration framework is going thru the every one of these documents. Lets consider that, the first documents which MongoDb scanned, has manufacturer as “Apple”. Then the mongoDb will check in the result set if there is any document with \_id as apple, if it is not avaible then it will create new document with \_id as the manufacturer i.e Apple and add new key num\_products whose value is added by 1 because of $sum operator. As there is no value to num\_products, then we it is set to 1 for the first run.

## While ruinng thru the products collections then mongoDb is also creating a collection to holds the results.

## If the second documents has a mufacturer as apple then it will only update the num\_products by 1 i.e 2.

## If the 3rd has a manuf as samsung and now we again look for document with \_id as samsung which is not there. So we simply create a new documents and num\_products key to 1.

## After the complete product documents are scanned and then a resultset of documents are created for a particulat stage, that gonna run thru the next stage of the pipeline. And that set of the documents is going to have a \_id equal to the manufacturer of each of the products.

**5) Compound Grouping :**

## Now we know how to group by single key, but what if we want to group by multiple keys?

## Example: if we want to group with respective to manufactures and categories.

## In SQL: select manufacturer, category, count(\*) from products order by manufacturer, category.

## IN mongo : we do the same in Aggregate framework using compound key. Rather than just say that we organise group by manufacturer, we can say that group bu manuf and category by specifying compound key. We can do it as below.

## \_id :{ “manufacturer” : “$manufacturer”, “Category”: “$category” }

## we can name the keys as our wish and so we can label them betterly as required.

## \_id:{“manufacturer”:”$manufacturer”}

## then the out of the resultset will be like \_id : “manufacturer : apple”

## Compound key can be any number of combination, not compulsory by 2.

**6) Document as ID:**

## the \_id of the document, can itself be a document. The only contains on \_id is that it should be a unique one.

## db.foo.insert( { \_id: {name:”pranith” , “role” : “ggg”}, data:[“a”,”b”] })

## this will be inserted into mognoDB.\_id should be unique . Ususally people wanted to keep it has a scalar value. But it is not compulsory.

## If we insert same about document, then it will throw a suplicate key exception.

**7) Aggragation Expressions:**

## Expression that can be used in group stage of the aggression piple line.

## $sum : it can be used to count thing , if we add 1 and if u give some value then it will sum that key.

## $avg: can give the avg of the key across the documents.

## $min: min value of key in a documents. This is as \_id that is specified.

## $max: max value of the key.

## $push : Only for arrays. We can push docuemnts into an array.

## $addToSet: Only for arrays. Same as push but takes cares of the uniqueness of the key.

## $first and $last: these gives the first and last value of the key. But before that the collection has to be sorted. Else then the values returned by these are arbitary.

**8) $sum :**

## We have already seen how to use sum to count the number of documents.

## Let us now understand, how sum is used to add all the value of a particular field say prices.

## db.products.aggregate([

## { $group :

## {

## \_id : {“manufacture” : “$manufacturer”},

## sum\_prises:{“$sum” : “$prices”}

## }

## 

## }

## ])

**8) $avg :** This is used to find the avarage of particular field with respect to particular \_id column.

Let us assume to get the average of price for a particular column.

## db.products.aggregate([

## { $group :

## {

## \_id : {“category “:”“$category”},

## avg\_prises{”$avg”: “$prices”}

## }

## 

## }

## ])

## 8) $addToSet :

## This is similar to listaggs. It will list all the value of the filed which is argument of $addToSet based on the group by \_id.

## db.products.aggregate([

## { $group :

## {

## \_id : {“manufacture” : “$manufacturer”},

## AllvaluesOfCategories:{“$addToSet” : “$categories”}

## }

## 

## }

## ])

## This will return a collection which include all the categories of a particulat manufacture.

## 9) $push :

## This is similar to listaggs. It will list all the value of the filed which is argument of $addToSet based on the group by \_id. Similar to $addToList but it does not gurantie adding of once or more. It add all.

## db.products.aggregate([

## { $group :

## {

## \_id : {“manufacture” : “$manufacturer”},

## AllvaluesOfCategoriesWithDuplicates:{“$push” : “$categories”}

## }

## 

## }

## ])

## This will return a collection which include all the categories of a particulat manufacture.

## 9) $Max and $min :

## Max and min value for a field based on the category field.

## db.products.aggregate([

## { $group :

## {

## \_id : {“manufacture” : “$manufacturer”},

## AllvaluesOfCategoriesWithDuplicates:{“$max” : “$price”}

## }

## 

## }

## ])

## 10) Double $group stages :

## One of the neat things of aggregration framework that’s even better than what you have available in sql is that you can run a particular aggregation stage more than a once.

## For instance, we can group more than once in the same aggregration query. This is called as double grouping.

db.fun.aggregate([{$group:{\_id:{a:"$a", b:"$b"}, c:{$max:"$c"}}}, {$group:{\_id:"$\_id.a", c:{$min:"$c"}}}])

## 11) $project :

## The next stage of the aggregation pipeline which we gonna discuss is $project phase. The project phase lets us to reshape the document as they come thru the pipeline.

## It’s a 1:1 stage of the pipeline, so for every document that comes into the project phase of the pipeline will leave the project phase.

## We can do the things like removing a key, adding new keys, reshape the keys(i.e taking a key and kepping into sub documents or wice versa), or use some simple functions on the the keys like $toLower, $toUpper, $add, $multiply.

## In the above query, instead of $group we are using $project. Aggregate function will uses a array of phases. So we can have grouping and projections and we could have multiple projections and multiple groupings. Right now, we are using only one stage.

db.products.aggregate([

{ $project :

{

\_id: 0,

‘makers’ : {$toLower : “$manufacturer”},

‘details’ : { ‘category’ : “$category”,

‘price’ : {“$multiply”: [“$price”,10] }

}

}

}

])

## First in the query is that we specified that we don’t want to include \_*id(*id:0) and then it could include the key “makers” which hold the lowercased value of the $manugacturer and then a sub-document as “details”. This sub document has keys category with value of category and then price whose value is equal to value of price multiple by 10.

## Example:

## Doc structure :

{

"city" : "ACMAR",

"loc" : [

-86.51557,

33.584132

],

"pop" : 6055,

"state" : "AL",

"\_id" : "35004"

## }

## Output:

{

"city" : "acmar",

"pop" : 6055,

"state" : "AL",

"zip" : "35004"

}

## Answer:

## db.zips.aggregate([{"$project" : {\_id: 0,city :{"$toLower" : "$city"},pop : 1,state:1 , 'zip' : "$\_id"}}])

## 12) $match :

## Match performs a filtering, which is a n:1 mapping. Match will go thru the each document and see if the document matches the criteria and if it does, then it will push it through the next stage of the aggrigation pipeline.

## We have 2 reason, why we have to check for a match.

## to filter out the documents and only perform aggregation on the subset on them.

## We might want to perform a match in order to filter the query itself. i.e first perform a aggregation and and then filter the result.

## Example:

## $match will take the argument a filter which is similar to find query. In the below example we are trying to find all the documents that whose value of state is CA.

## db.zips.aggregate([{$match : { state : “CA”} }])

## Lets us now try to fetch the population of each city by grouping on the city.

## db.zips.aggregate([{$match : { state : “CA”} } , { $group : { *id: “$city” , population: {“$sum” : “$pop” } , zip*codes : {“$addToSet” : “$\_id”} } } ])

## This is how we can use multiple stage in aggregate function. $match stage will filter the documents and then $group stage will group based on the city and by summing all the population and building the set of zipcodes.

## Also we can reshape using project state.

## db.zips.aggregate([

## {$match : { state : “CA”} } ,

## {$group : { \_*id: “$city” , population: {“$sum” : “$pop” } , zip*codes : {“$addToSet” : “$\_id”} }} ,

## {$project : {\_id:0 , city:”$\_id”, population : 1, zipcodes : 1 } }

## ])

## One thing to note about $match (and $sort) is that they can use indexes, but only if done at the beginning of the aggregation pipeline.

## 13) $sort :

## There are few thing we have to know about sorting.

## 1) The aggregation framework supports both disk and memory based sorting. By Default the aggregation framework will try to sort in memory and there is a limit of 100mb for any given pipeline stage unless we allow the use of disk in aggregation. we can set that option in aggregation framework. If we don’t set that option, we gonna wind up with the memory based sort and all the pipeline stages or any particular pipeline stage is limited to 100MB. And there is an option to do a disk-based sort which you might wanna use if there’s a lot of data that’s gonna be produced in the pipeline stages.

## 2) sorting can be done before or after the grouping stage based on the requirments.

## Example:

db.zips.aggregate([

{$match : { state : “CA”} } ,

{$group : { \_*id: “$city” , population: {“$sum” : “$pop” } , zip*codes : {“$addToSet” : “$\_id”} }} ,

{$project : {\_id:0 , city:”$\_id”, population : 1, zipcodes : 1 } },

{$sort : { population : -1} }

])

## db.zips.aggregate([ { $sort : { "state" :1 ,"city": 1 } } ])

## 14) $skip and $limit :

## Remember sort has to be done first and then skip and then limit. In other case, result is undefined.

## If we recall, for find query, the order of sort, skip and limit in the query does not matter beecause server will take care of the order. But in the case of aggregate framework, we are giving a ordered list of stages, so the order in which we specify skip and limit does matters.

db.zips.aggregate([

{$match:{state:"NY"}},

{$group:{\_id: "$city",population: {$sum:"$pop"},}},

{$project:{\_id: 0,city: "$\_id",population: 1,}},

{$sort:{population:-1}},

{$limit: 5},

{$skip: 10}

])

## The above query will give 0 docs because we are limiting the size to 5 and then skipping it to 10. The revesing will give anser as below query.

db.zips.aggregate([

{$match:{state:"NY"}},

{$group:{\_id: "$city",population: {$sum:"$pop"},}},

{$project:{\_id: 0,city: "$\_id",population: 1,}},

{$sort:{population:-1}},

{$skip: 10} ,{$limit: 5}

])

## 15) $first and $last : First and last element of the field in the result set of the pipeline.

{ "\_id" : 0, "a" : 0, "b" : 0, "c" : 21 }

{ "\_id" : 1, "a" : 0, "b" : 0, "c" : 54 }

{ "\_id" : 2, "a" : 0, "b" : 1, "c" : 52 }

{ "\_id" : 3, "a" : 0, "b" : 1, "c" : 17 }

{ "\_id" : 4, "a" : 1, "b" : 0, "c" : 22 }

{ "\_id" : 5, "a" : 1, "b" : 0, "c" : 5 }

{ "\_id" : 6, "a" : 1, "b" : 1, "c" : 87 }

{ "\_id" : 7, "a" : 1, "b" : 1, "c" : 97 }

db.fun.aggregate([

{$match:{a:0}},

{$sort:{c:-1}},

{$group:{\_id:"$a", c:{$first:"$c"}}}

])

## 16) Using $unwind :

## In MongoDb, Documents can have arrays and it is not easy to group something on array, unless we it out of the array form and make it more flat. Arrays are kind of prejoined data.

## when we have a documents like

## {a: 1, b:2 , c: [ “apple”, “mongo” , “orange”] };

## now, we are join to use $unwind to unjoin the data and then basically rejoin it in a way that lets us do grouping calculation on it.

## Lets say, we have unwind on c. Then for each doc, it’s going to create a document with the remainder of the doc plus, each value from this array.

## {a: 1, b:2 , c: “apple”};

## {a: 1, b:2 , c: “mongo” ] };

## {a: 1, b:2 , c: “orange” };

## So we got 3 docs when we unwind the docs on c. and once I do this unwind, then we can use any aggregation expression.

## Thats typically the way we ill use it. And if there are 5 docs for example, and each one has 3 elements in array, then after unwind we will get 15 docs out of unwind stage. So this is sort of data explosion from unwind, but we need that bcz we need data, that looks honestly, a little more tabular, a little bit relational and then we can group it on the element in the array..

## 17) Using $unwind :

db.posts.aggregate([

{“$unwind” : $tags},

{ “$group” : { “\_id” : “$tags” , “count”:{ “$sum” : 1} } },

{ “$sort” : {“count” : -1} },

{ “$limit” : 10 },

{“$project” : { \_id : 0, ‘tag’:”$\_tag”, count : 1}}

## ])

## Quiz: $unwind example

## Which grouping operator will enable to you to reverse the effects of an unwind?

## a) $sum

## b) $addToSet

c) $push

d) $first

## 18) double $unwind :

## Sometimes we may have more then one array in a documents and in that case we may want to double unwind the array. Therefore creating a cortical product of 2 arrays and rest of the document.Let us consider the below collection of cloths with different sizes and colors:

## 

db.inventory.aggregate([

{“$unwind”:”$sizes”},

{“$unwind”:”$colors”},

{“$group”:{

‘\_id’ : { ‘size’:’$size’ , ‘color’ : ‘$colots’},

‘count’:{‘$sum’ : 1}

}

},

## ])

## 19) Mapping between SQL and Aggregration :

## The [aggregation pipeline](https://docs.mongodb.org/manual/core/aggregation-pipeline/) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

## The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB [aggregation operators](https://docs.mongodb.org/manual/reference/operator/aggregation/" \l "aggregation-pipeline-operator-reference):

| SQL Terms, Functions, and Concepts | MongoDB Aggregation Operators |
| --- | --- |
| WHERE | [$match](https://docs.mongodb.org/manual/reference/operator/aggregation/match/" \l "pipe._S_match" \o "$match) |
| GROUP BY | [$group](https://docs.mongodb.org/manual/reference/operator/aggregation/group/" \l "pipe._S_group" \o "$group) |
| HAVING | [$match](https://docs.mongodb.org/manual/reference/operator/aggregation/match/" \l "pipe._S_match" \o "$match) |
| SELECT | [$project](https://docs.mongodb.org/manual/reference/operator/aggregation/project/" \l "pipe._S_project" \o "$project) |
| ORDER BY | [$sort](https://docs.mongodb.org/manual/reference/operator/aggregation/sort/" \l "pipe._S_sort" \o "$sort) |
| LIMIT | [$limit](https://docs.mongodb.org/manual/reference/operator/aggregation/limit/#pipe._S_limit) |
| SUM() | [$sum](https://docs.mongodb.org/manual/reference/operator/aggregation/sum/#grp._S_sum) |
| COUNT() | [$sum](https://docs.mongodb.org/manual/reference/operator/aggregation/sum/#grp._S_sum) |
| join | No direct corresponding operator; however, the [$unwind](https://docs.mongodb.org/manual/reference/operator/aggregation/unwind/#pipe._S_unwind)operator allows for somewhat similar functionality, but with fields embedded within the document. |

**19) Some Common SQL example**

## The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

## The SQL examples assume two tables, orders and order\_lineitem that join by theorder\_lineitem.order\_id and the orders.id columns.

## The MongoDB examples assume one collection orders that contain documents of the following prototype:

## { cust\_id: "abc123", ord\_date: ISODate("2012-11-02T17:04:11.102Z"), status: 'A', price: 50, items: [ { sku: "xxx", qty: 25, price: 1 }, { sku: "yyy", qty: 25, price: 1 } ] }

| SQL Example | MongoDB Example | Description |
| --- | --- | --- |
| SELECT COUNT(\*) AS count FROM orders | db.orders.aggregate( [ { $group: { \_id: null, count: { $sum: 1 } } } ] ) | Count all records fromorders |
| SELECT SUM(price) AS total FROM orders | db.orders.aggregate( [ { $group: { \_id: null, total: { $sum: "$price" } } } ] ) | Sum theprice field from orders |
| SELECT cust\_id, SUM(price) AS total FROM orders GROUP BY cust\_id | db.orders.aggregate( [ { $group: { \_id: "$cust\_id", total: { $sum: "$price" } } } ] ) | For each uniquecust\_id, sum theprice field. |
| SELECT cust\_id, SUM(price) AS total FROM orders GROUP BY cust\_id ORDER BY total | db.orders.aggregate( [ { $group: { \_id: "$cust\_id", total: { $sum: "$price" } } }, { $sort: { total: 1 } } ] ) | For each uniquecust\_id, sum theprice field, results sorted by sum. |
| SELECT cust\_id, ord\_date, SUM(price) AS total FROM orders GROUP BY cust\_id, ord\_date | db.orders.aggregate( [ { $group: { \_id: { cust\_id: "$cust\_id", ord\_date: { month: { $month: "$ord\_date" }, day: { $dayOfMonth: "$ord\_date" }, year: { $year: "$ord\_date"} } }, total: { $sum: "$price" } } } ] ) | For each uniquecust\_id,ord\_dategrouping, sum the pricefield. Excludes the time portion of the date. |
| SELECT cust\_id, count(\*) FROM orders GROUP BY cust\_id HAVING count(\*) > 1 | db.orders.aggregate( [ { $group: { \_id: "$cust\_id", count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } } ] ) | For cust\_idwith multiple records, return thecust\_id and the corresponding record count. |
| SELECT cust\_id, ord\_date, SUM(price) AS total FROM orders GROUP BY cust\_id, ord\_date HAVING total > 250 | db.orders.aggregate( [ { $group: { \_id: { cust\_id: "$cust\_id", ord\_date: { month: { $month: "$ord\_date" }, day: { $dayOfMonth: "$ord\_date" }, year: { $year: "$ord\_date"} } }, total: { $sum: "$price" } } }, { $match: { total: { $gt: 250 } } } ] ) | For each uniquecust\_id,ord\_dategrouping, sum the pricefield and return only where the sum is greater than 250. Excludes the time portion of the date. |
| SELECT cust\_id, SUM(price) as total FROM orders WHERE status = 'A' GROUP BY cust\_id | db.orders.aggregate( [ { $match: { status: 'A' } }, { $group: { \_id: "$cust\_id", total: { $sum: "$price" } } } ] ) | For each uniquecust\_idwith status A, sum theprice field. |
| SELECT cust\_id, SUM(price) as total FROM orders WHERE status = 'A' GROUP BY cust\_id HAVING total > 250 | db.orders.aggregate( [ { $match: { status: 'A' } }, { $group: { \_id: "$cust\_id", total: { $sum: "$price" } } }, { $match: { total: { $gt: 250 } } } ] ) | For each uniquecust\_idwith status A, sum theprice field and return only where the sum is greater than 250. |
| SELECT cust\_id, SUM(li.qty) as qty FROM orders o, order\_lineitem li WHERE li.order\_id = o.id GROUP BY cust\_id | db.orders.aggregate( [ { $unwind: "$items" }, { $group: { \_id: "$cust\_id", qty: { $sum: "$items.qty" } } } ] ) | For each uniquecust\_id, sum the corresponding line item qtyfields associated with the orders. |
| SELECT COUNT(\*) FROM (SELECT cust\_id, ord\_date FROM orders GROUP BY cust\_id, ord\_date) as DerivedTable | db.orders.aggregate( [ { $group: { \_id: { cust\_id: "$cust\_id", ord\_date: { month: { $month: "$ord\_date" }, day: { $dayOfMonth: "$ord\_date" }, year: { $year: "$ord\_date"} } } } }, { $group: { \_id: null, count: { $sum: 1 } } } ] ) | Count the number of distinctcust\_id,ord\_dategroupings. Excludes the time portion of the date. |

## 20 )Limitation of Aggregation Framework:

## we gonna now discuss about the limitation in aggregation framework and how can be get around them and what you should think about when you;re using it.

## 1) By Default, we have 100mb limit on memory for pipeline stages. We can overcome this by using allowDiskUse option, which will get you around 100mb limit. But unless we specify that option to aggregation, we will be limited to that 100mb and we will find that the queries will not some back, if they re very large, and they have large intermediary results.

## 2) if we wanted to return the result in one signle documents then there is 16mb size limit on docs. And since photon does return the results in one documents there’s 16mb limit by default in python. We can simple come out of it by using a cursor.

## 3) In a sharded system, as soon as we use a group by or a sort or anything that requires looking at all results, then we may find the aggregation of a large collection on a shared cluster may not be quite as good as expected. To this, hadoop and map reduce and we can go do that using hadoop connector. And there is also other map reduce functionality in MongoDb but has several limitations. Not recommended unless for simple stuff . Use aggregation instead of map-reduce.

## 20 )Aggregation framework with java Drivers.

## We have to user aggregate funtion and should pass of documents as params.

## Example 1

MongoClient client = **new** MongoClient();

MongoDatabase database = client.getDatabase("m101");

MongoCollection<Document> collection = database.getCollection("zips");

Document document = **new** Document().append(

"$group",

**new** Document().append("\_id", "$state").append("PopulatonCount",

**new** Document().append("$sum", "$pop")));

List<Document> pipeline = Arrays.*asList*(document);

MongoCursor<Document> cursor = collection.aggregate(pipeline)

.iterator();

**try** {

**while** (cursor.hasNext()) {

Document docs = cursor.next();

System.***out***.println(docs.get("\_id"));

}

} **finally** {

cursor.close();

}

client.close();

## Using Document parse to parse a json string to document object.

MongoClient client = **new** MongoClient();

MongoDatabase database = client.getDatabase("m101");

MongoCollection<Document> collection = database.getCollection("zips");

Document document1 = Document.*parse*("{$group:{\_id :\"$state\",count:{$sum:\"$pop\"}}}");

List<Document> pipeline = Arrays.*asList*(document1);

MongoCursor<Document> cursor = collection.aggregate(pipeline)

.iterator();

**try** {

**while** (cursor.hasNext()) {

Document docs = cursor.next();

System.***out***.println(docs.get("\_id"));

}

} **finally** {

cursor.close();

}

client.close();