Kenny R N Wilson

MongoDB

Introduction

THIS DOCUMENT COVERS

Cheat Sheet

General

	Column Header
Binaries	`C:\Program Files\MongoDB\Server\4.4\bin`
Default End Point	`localhost:27017`
Dump	`mongodump --uri="mongodb://localhost:27017" --archive=myarchive --gzip`
Restore	`mongorestore --uri="mongodb://localhost:27017" --drop --archive=myarchive --gzip`

Basic Commands

	Column Header
Create database	`use mydb;`
Create collection	`db.createCollection("c")`
List collection names	`show collections`
List databases	`show dbs`
Insert one doc	`db.c.insertOne( { nm: "Jo"} );`

Insert many docs	`db.cl.insertMany([`
	`    {name:"jim", age: 31},`
	`    {name:"sam", age: 22},`
	`])`

Get all docs	`db.c.find()`
Drop collections	`db.c.drop();`
Delete single document	`db.c.deleteOne( {nm:"Joe"})`

Basic Queries

	Column Header
Embedded Document	`find({“person.sex”:"male"}, {})`

Update

Increment

$inc can be used to increment or decrement fields of type integer, long, double, decimal.

Input collection

```
{
    "name" : "john",
    "age" : 32.0
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$inc": {"age": 1}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0
}
```

Set

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$set": {"sex": “male”}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male"
}
```

Arrays

Adding Single Values to An Array

$push can be used to add an element to an array field. If the array field does not exist it will be created, and the value added.

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male"
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$push": {"values": 100}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0
    ]
}
```

ADDING MULTIPLE VALUES TO AN ARRAY

Input collection

```
  {
      "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
      "name" : "john",
      "age" : 33.0,
      "sex" : "male",
      "values" : [
          100.0
      ]
}
```

Update

```
  db.cl.updateOne(
      {"name":"john"},
      {"$push": {"values": { "$each" : [ 200,300,400]}}}
  )
```

Result

```
  {
      "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
      "name" : "john",
      "age" : 33.0,
      "sex" : "male",
      "values" : [
          100.0,
          200.0,
          300.0,
          400.0
      ]
  }
```

Add Single Value To Set

Treats the array as a set in that it will not add duplicates.

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0,
        200.0,
        300.0,
        400.0
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$addToSet": {"values": 400}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0,
        200.0,
        300.0,
        400.0
    ]
}
```

> **Note:**
>
> We can add multiple set values by using $each in the same way we use it with $push

Removing Elements From End of Array

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0,
        200.0,
        300.0,
        400.0,
        410.0
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$pop": {"values": 1}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0,
        200.0,
        300.0,
        400.0,
    ]
}
```

Removing Elements From Start of Array

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        100.0,
        200.0,
        300.0,
        400.0,
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$pop": {"values": -1}}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        200.0,
        300.0,
        400.0,
    ]
}
```

Removing Matching Elements From Array

Input collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        200.0,
        300.0,
        400.0,
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$pull": {"values": 300}}
)
```

Result

```
   "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        200.0,
        400.0,
    ]
}
```

Updating array elements by index

Input Collection

```
     "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
      "name" : "john",
      "age" : 33.0,
      "sex" : "male",
      "values" : [
          200.0,
          400.0,
      ]
  }
```

Update

```
  db.cl.updateOne(
      {"name":"john"},
      {"$inc": {"values.0": 1}}
  )
```

Result

```
  {
      "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
      "name" : "john",
      "age" : 33.0,
      "sex" : "male",
      "values" : [
          201.0,
          400.0
      ]
  }
```

Updating array elements by $ operator

Input Collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        201.0,
        400.0
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john", "values": 201},
    {"$set": {"values.$": 203 }}
)
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        203.0,
        400.0
    ]
}
```

ARRAY FILTERS

Enables us to update array elements that match a particular filter

Input Collection

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        203.0,
        400.0
    ]
}
```

Update

```
db.cl.updateOne(
    {"name":"john"},
    {"$set": {"values.$[elem]": 205 }},
    {
        arrayFilters: [{"elem": { $lte: 205}}]
    }
))
```

Result

```
{
    "_id" : ObjectId("5faad5920615ba4bb7e869c7"),
    "name" : "john",
    "age" : 33.0,
    "sex" : "male",
    "values" : [
        205.0,
        400.0
    ]
}
```

Upserts

Upsert creates the doc if it doesn't exist and updates it if it does. The values of the created document are taken from the criteria

Input Collection

```
{
    "_id" : ObjectId("5faad6370615ba4bb7e869c8"),
    "name" : "jim",
    "age" : 31.0
}
```

Update

```
db.cl.updateOne(
    {"name":"dave"},
    {"$set" : {age: 45}},
    {"upsert": true}
)
```

Result

```
{
    "_id" : ObjectId("5faad6370615ba4bb7e869c8"),
    "name" : "jim",
    "age" : 31.0
}
{
    "_id" : ObjectId("5fab9c1ef70513dae1565877"),
    "name" : "dave",
    "age" : 45.0
}
```

UpdateMany

Input Collection

```
{ "name" : "jim","age" : 31.0 }
{ "name" : "sam","age" : 22.0}
{ "name" : "dave","age" : 45.0 }
```

Update

```
db.cl.updateMany({},
    {"$set" : {visited: true}}
)
```

Result

```
{ "name" : "jim","age" : 31.0, "visited" : true }
{ "name" : "sam","age" : 22.0, "visited" : true}
{ "name" : "dave","age" : 45.0, "visited" : true }
```

Returning the documents updated

Allow us to atomically get the value of modified document using findOneAndDelete, findOneAndModify and findOneAndUpdate. The modified document is returned. A flag controls if the version before or after is returned.

Queries

Embedded Documents

Match Whole Document

This will only work if we don't add to the subdocument

Input Collection

```
  {
      "_id" : ObjectId("5fabc0d64b8b4ca47aa23e36"),
      "subdoc" : {
          "first" : "kenny",
          "second" : "wilson"
      }
  }
```

Query

```
  db.embde.find({subdoc: {first: "kenny", second: "wilson"}})
```

Results

```
  {
      "_id" : ObjectId("5fabc0d64b8b4ca47aa23e36"),
      "subdoc" : {
          "first" : "kenny",
          "second" : "wilson"
      }
}
```

Dot Notation

This will work if the collection changes.

Input Collection

```
  {
      "_id" : ObjectId("5fabc0d64b8b4ca47aa23e36"),
      "subdoc" : {
          "first" : "kenny",
          "second" : "wilson"
      }
  }
```

Query

```
  b.embde.find({"subdoc.first" : "kenny", "subdoc.second" : "wilson"})
```

Results

```
  {
      "_id" : ObjectId("5fabc0d64b8b4ca47aa23e36"),
      "subdoc" : {
          "first" : "kenny",
          "second" : "wilson"
      }
}
```

And Filter

The following only return male whose name is don.

Input collections

```
{ "nm" : "ken", "age" : 45 }
{ "nm" : "kim", "age" : 23 }
{ "nm" : "don", "age" : 60, "sex" : "male }
```

Query

```
db.c.find({sex:"male", nm:"don"}, {nm:1,_id:0})
```

Result

```
{ "nm" : "don" }
```

Logical OR

We can list anyone who is 23 or whose name is jon as follows

Input Collection

```
{ "nm" : "ken", "age" : 45 }
{ "nm" : "kim", "age" : 23 }
{ "nm" : "jon", "age" : 60 }
```

Query

```
db.c.find({ "$or" : [  {age:23}, {nm: "jon"}]}, {_id:0})
```

Result

```
{ "nm" : "kim", "age" : 23 }
{ "nm" : "jon", "age" : 60 }
```

Not equal

Input Collection

```
  { "name" : "jim","age" : 31.0 }
  { "name" : "sam","age" : 22.0}
  { "name" : "dave","age" : 45.0 }
```

Query

```
db.cl.find({name: {"$ne": "sam"}}, {_id:0})
```

Result

```
  { "name" : "jim","age" : 31.0 }
  { "nm" : "jon", "age" : 60 }
```

Not

Not is a meta operator that is applied to other criteria or regular expressions.

Input Collection

```
{ "name" : "jim","age" : 31.0 }
{ "name" : "sam","age" : 22.0}
{ "name" : "dave","age" : 45.0 }
```

Query

```
db.cl.find({name: {"$not" : {"$in" : ["sam", "jim"]}}})
```

Result

```
{ "name" : "dave","age" : 45.0 }
```

Conditionals

We can list all the people whose age is between 21 and 50 inclusive as follows. Conditionals supported include

- "$lt"
- "$lte"
- "$gt"
- "$gte"

Input collections

```
{ "nm" : "ken", "age" : 45 }
{ "nm" : "kim", "age" : 23 }
{ "nm" : "jon", "age" : 60 }
```

Query

```
db.c.find({ age: {"$gte" : 23, "$lte" : 50}}, {_id:0})
```

Result

```
{ "nm" : "ken", "age" : 45 }
{ "nm" : "kim", "age" : 23 }
```

In

We can list the 23- and 45-years old.

Input collections

```
  { "nm" : "ken", "age" : 45 }
  { "nm" : "kim", "age" : 23 }
  { "nm" : "jon", "age" : 60 }
```

Query

```
  db.c.find({ age: {"$in" : [23,45]}}, {_id:0})
```

Result

```
  { "nm" : "ken", "age" : 45 }
  { "nm" : "kim", "age" : 23 }
```

Null

Consider the following

Input Collection

```
{ "x" : 1, "nm" : null }
{ "x" : 2 }
{ "x" : 3, "nm" : "ken" }
```

Then checking for null returns the documents that have null for that field value or for which the field is not set at all

Query

```
db.c.find({nm:null}, {_id:0})
```

Results

```
{ "x" : 1, "nm" : null }
{ "x" : 2 }
```

Null and exists

If we want to check for where a field actually exists and is null we do

Input Collection

```
{ "x" : 1, "nm" : null }
{ "x" : 2 }
{ "x" : 3, "nm" : "ken" }
```

Query

```
db.c.find({nm: {"$eq" :null, "$exists":true}}, {_id:0})
```

Results

```
{ "x" : 1, "nm" : null }
```

Regular Expressions

Input Collection

```
{
    "_id" : ObjectId("5faad6370615ba4bb7e869c8"),
    "name" : "jim",
    "age" : 31.0,
    "visited" : true
}

{
    "_id" : ObjectId("5faad6370615ba4bb7e869c9"),
    "name" : "sam",
    "age" : 22.0,
    "visited" : true
}

{
    "_id" : ObjectId("5fab9c1ef70513dae1565877"),
    "name" : "dave",
    "age" : 45.0,
    "visited" : true
}
```

Query

```
db.cl.find({name: {"$regex" : /AVE/i}})
```

Results

```
{
    "_id" : ObjectId("5fab9c1ef70513dae1565877"),
    "name" : "dave",
    "age" : 45.0,
    "visited" : true
}
```

Searching Arrays

Contains Single Specified Value

Image we have this

```
{ "fruit" : [ "apple", "orange", "pear" ] }
{ "fruit" : [ "apple", "banana" ] }
{ "fruit" : [ "banana", "kiwi" ] }
```

Then we can find all documents whose fruit array contains apple as follows

```
db.c1.find({fruit: "apple"},{_id:0})
```

Which gives us

```
{ "fruit" : [ "apple", "orange", "pear" ] }
{ "fruit" : [ "apple", "banana" ] }
```

Contains All Specified Values

Image we have this

```
{ "fruit" : [ "apple", "orange", "pear" ] }
{ "fruit" : [ "apple", "banana" ] }
{ "fruit" : [ "banana", "kiwi" ] }
```

We can find all documents whose fruit field’s array contains both the values apple and pear

```
> db.c1.find({fruit: {"$all" : [ "apple", "pear"]}},{_id:0})
```

Which gives us

```
{ "fruit" : [ "apple", "orange", "pear" ] }
```

Arrays of Specified Size

We can query for all documents whose fields array value is of a certain length

```
db.fruit.find({fruit: {"$size":2}} , {_id:0})
{ "fruit" : [ "apple", "pear" ] }
```

Imagine we have this

```
{ "fruit" : [ "apple", "orange", "pear" ] }
{ "fruit" : [ "apple", "banana" ] }
{ "fruit" : [ "banana", "kiwi" ] }
```

We can find all documents whose fruit field is an array of size 2

```
db.c1.find({fruit: {"$size":2} },{_id:0})
```

Which gives us

```
{ "fruit" : [ "apple", "banana" ] }
{ "fruit" : [ "banana", "kiwi" ] }
```

Slice

A positive number returns first n elements so given

```
{ "x" : [ 10, 11, 12, 13, 14, 15, 15 ] }
```

We can take first 3 elements in the result

```
db.nums.find({}, {x:{"$slice":3}})
```

Giving

```
  { "_id" : ObjectId("5ec25a6b9d4450d02edd4142"), "x" : [ 10, 11, 12 ] }
```

A negative number returns last three

```
db.nums.find({}, {x:{"$slice":-3}})
```

Giving

```
  { "_id" : ObjectId("5ec25a6b9d4450d02edd4142"), "x" : [ 14, 15, 16 ] }
```

Indexing

We can index into arrays

```
  db.nums.find({"x.1":11})
```

ARRAY EQUALITY

If we want to perform an exact match, we can use array equality. Imagine we have this

```
{ "fruit" : [ "apple", "orange", "pear" ] }
{ "fruit" : [ "apple", "banana" ] }
{ "fruit" : [ "banana", "kiwi" ] }
```

We can do an exact match as follows.

```
db.c1.find({fruit: ["banana", "kiwi"]},{_id:0})
```

```
{ "fruit" : [ "banana", "kiwi" ] }
```

Which gives us

```
{ "fruit" : [ "apple", "pear" ] }
```

Match Single Element and Return

Input collections

```
{
    "_id" : ObjectId("5fabb227f70513dae1565f80"),
    "name" : "Kenny",
    "arr" : [
        {
            "key" : "A",
            "value" : 10
        },
        {
            "key" : "B",
            "value" : 20
        }
    ]
}
```

Query

```
db.cl2.find({"arr.key" : "B"}, {"_id" : 0, "arr.$" : 1})
```

Result

```
{
    "arr" : [
        {
            "key" : "B",
            "value" : 20
        }
    ]
}
```

Where

If all else fails, we can use where to provide custom filter logic.

```
db.embde.find({"$where" : function () {
    return this.subdoc.first == "kenny";
}});
```

This should be avoided as it comes at a high performance cost.

Aggregation

Aggregation is carried out as a pipeline. Each stage takes in the pipeline is a data processor that takes a stream of documents as input and produces a stream of input as output. The stages have parameters that provide control.

https://docs.mongodb.com/manual/meta/aggregation-quick-reference/

Filtration/match ($match)

Input collection

```
{
    "name" : "kenny",
    "age" : 45
}
{
    "name" : "john",
    "age" : 40
}
{
    "name" : "dave",
    "age" : 30
}
```

Pipeline

```
db.ag.aggregate(
    [
        {$match : {name: "john"}}
    ]
)
```

Result

```
{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24723"),
    "name" : "john",
    "age" : 40
}
```

Sorting ($sort)

Input collection

```
{
    "name" : "kenny",
    "age" : 45
}
{
    "name" : "john",
    "age" : 40
}
{
    "name" : "dave",
    "age" : 30
}
```

Pipeline

```
db.ag.aggregate([
    {$sort: {name:1}}
]
)
```

Result

```
{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24726"),
    "name" : "dave",
    "age" : 30
}

{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24723"),
    "name" : "john",
    "age" : 40
}

{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24720"),
    "name" : "kenny",
    "age" : 45
}
```

Skip ($skip)

Input collection

```
{
    "name" : "kenny",
    "age" : 45
}
{
    "name" : "john",
    "age" : 40
}
{
    "name" : "dave",
    "age" : 30
}
```

Pipeline

```
db.ag.aggregate([
    {$sort: {name:1}},
    {$skip: 2}
]
)
```

Result

```
{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24720"),
    "name" : "kenny",
    "age" : 45
}
```

Limit ($limit)

Input collection

```
{
    "name" : "kenny",
    "age" : 45
}
{
    "name" : "john",
    "age" : 40
}
{
    "name" : "dave",
    "age" : 30
}
```

Pipeline

```
db.ag.aggregate([
    {$sort: {name:1}},
    {$limit: 2}
]
)
```

Result

```
{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24726"),
    "name" : "dave",
    "age" : 30
}

{
    "_id" : ObjectId("5fabfdd54b8b4ca47aa24723"),
    "name" : "john",
    "age" : 40
}
```

Projecting ($project)

Simple

Note the use of $ to specify the value of the name field which we project to a field called firstName.

Input collection

```
{
    "name" : "kenny",
    "age" : 45
}
{
    "name" : "john",
    "age" : 40
}
{
    "name" : "dave",
    "age" : 30
}
```

Pipeline

```
db.ag.aggregate(
    [
        {$project :
            {
                _id:0,
                firstName: "$name"
            }
         }
    ]
)
```

Result

```
{
    "firstName" : "kenny"
}
{
    "firstName" : "john"
}

{
    "firstName" : "dave"
}
```

Unwinding Arrays

We can unwind to flatten arrays. We take one document and create three results

Input collection

```
{
    "_id" : ObjectId("5faaa0eef70513dae155ffcf"),
    "firstname" : "kenny",
    "secondname" : "wilson",
    "values" : [
        100,
        200,
        300
    ]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {$unwind: "$values"},
    {$project: {
        _id: 0,
        firstname : 1,
        secondname: 1,
        value: "$values"
        }
    }
])
```

Result

```
/* 1 */
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "value" : 100
}

/* 2 */
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "value" : 200
}

/* 3 */
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "value" : 300
}
```

Array Expressions

Filter

Select a subset of array values based on a condition. Notice the use of double dollar $$ to specify a variable in our expression.

Input collection

```
{
    "_id" : ObjectId("5faaa0eef70513dae155ffcf"),
    "firstname" : "kenny",
    "secondname" : "wilson",
    "values" : [100, 200, 300]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {
        $project:
        {
            _id:0,
            secondname:1,
            vals:
            {
                $filter:
                {
                    input: "$values",
                    as: "val",
                    cond: {$gte : ["$$val", 200]}
                }
            }
        }
    }
])
```

Result

```
{
    "secondname" : "wilson",
    "vals" : [
        200,
        300
    ]
}
```

ARRAYELEMAT

Specify particular elements within an array

Input collection

```
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "age" : 46,
    "values" : [
        100,
        50,
        200,
        500,
        300
    ]
}

{
    "firstname" : "jari",
    "secondname" : "litmanen",
    "age" : 35,
    "values" : [
        10,
        5,
        210,
        490,
        310
    ]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {
        $project:
        {
            _id:0,
            secondname:1,
            ar_first : {$arrayElemAt : ["$values", 0]},
            ar_last : {$arrayElemAt : ["$values", -1]}
        }
    }
])
```

Result

```
{
    "secondname" : "wilson",
    "ar_first" : 100,
    "ar_last" : 300
}

{
    "secondname" : "lit     manen",
    "ar_first" : 10,
    "ar_last" : 310
}
```

Slice

Input collection

```
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "age" : 46,
    "values" : [
        100,
        50,
        200,
        500,
        300
    ]
}

{
    "firstname" : "jari",
    "secondname" : "litmanen",
    "age" : 35,
    "values" : [
        10,
        5,
        210,
        490,
        310
    ]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {
        $project:
        {
            _id:0,
            secondname:1,
            first_two : {$slice : ["$values", 0,2]}
        }
    }
])
```

Result

```
{
    "secondname" : "wilson",
    "first_two" : [
        100,
        50
    ]
}

{
    "secondname" : "litmanen",
    "first_two" : [
        10,
        5
    ]
}
```

SIZE

Input collection

```
{
    "firstname" : "kenny",
    "secondname" : "wilson",
    "age" : 46,
    "values" : [
        100,
        50,
        200,
        500,
        300
    ]
}

{
    "firstname" : "jari",
    "secondname" : "litmanen",
    "age" : 35,
    "values" : [
        10,
        5,
        210,
        490,
        310
    ]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {
        $project:
        {
            _id:0,
            secondname:1,
            arr_size : {$size : "$values"}
        }
    }
])
```

Result

```
{
    "secondname" : "wilson",
    "arr_size" : 5
}

{
    "secondname" : "litmanen",
    "arr_size" : 5
}
```

Accumulators

PROJECT PHASE

When used in the project phase accumulators must operate on arrays within a single document.

Input collection

```
{
    "_id" : ObjectId("5faaa0eef70513dae155ffcf"),
    "firstname" : "kenny",
    "secondname" : "wilson",
    "age" : 46,
    "values" : [100, 50, 200, 500, 300    ]
}
```

Pipeline

```
db.unwindexamples.aggregate([
    {
        $project:
        {
            _id:0,
            secondname:1,
            minimum: { $min: "$values"},
            maximum: { $max: "$values"},
            first:   { $first: "$values"},
            last:   { $last: "$values"},
            sum:   { $sum: "$values"},
            mean:   { $avg: "$values"},

        }
    }
])
```

Result

```
{
    "secondname" : "wilson",
    "minimum" : 50,
    "maximum" : 500,
    "first" : 100,
    "last" : 300,
    "sum" : 1150,
    "mean" : 230.0
}
```

Group Phase

Avg

Input collection

```
{
    "year" : 2019,
    "profit" : 20000
}

{
    "year" : 2019,
    "profit" : 10000
}

{
    "year" : 2019,
    "profit" : 5000
}

{
    "year" : 2020,
    "profit" : 2000
}

{
    "year" : 2020,
    "profit" : 100
}
```

Pipeline

```
db.grouping.aggregate([
    {
        $group : {
            _id: {year: "$year"},
            avg_prof: {$avg: "$profit"}
        },
    },
])
```

Results

```
{
    "_id" : {
        "year" : 2020
    },
    "avg_prof" : 1050.0
}

{
    "_id" : {
        "year" : 2019
    },
    "avg_prof" : 11666.6666666667
}
```

Sum

Input collection

```
  {
      "year" : 2019,
      "profit" : 20000
  }

  {
      "year" : 2019,
      "profit" : 10000
  }

  {
      "year" : 2019,
      "profit" : 5000
  }

  {
      "year" : 2020,
      "profit" : 2000
  }

  {
      "year" : 2020,
      "profit" : 100
  }
```

Pipeline

```
  db.grouping.aggregate([
      {
          $group : {
              _id: {year: "$year"},
              sum: {$sum: "$profit"}
          },
      },
  ])
```

Results

```
{
      "_id" : {
          "year" : 2020
      },
      "sum" : 2100
  }

  {
      "_id" : {
          "year" : 2019
      },
      "sum" : 35000
  }
}
```

Fault Tolerance

Replica Sets

Replication is the process of duplicating data on multiple machines to increase fault tolerance. In MongoDB replication is provided by a replica set which is a set of MongoDB servers. Within a replica set one server is designated the primary server and all other servers are secondary. All writes go through the primary server. Should the primary server crash the secondary severs work together to elect a new primary.

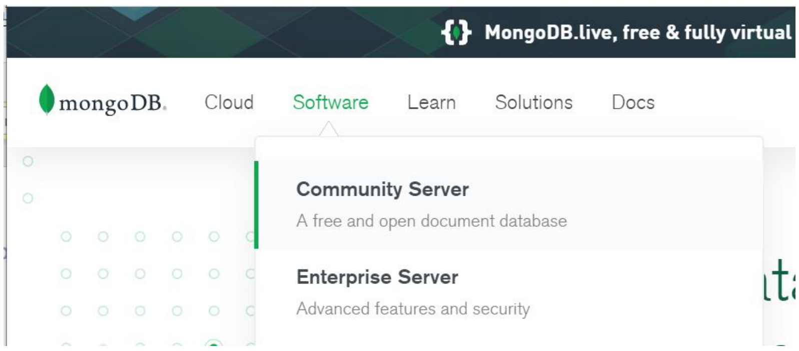
In addition to fault tolerance configuring a replica set can also increase read performance as read requests are distributed across the secondary nodes in the replica set.

Sharding

Sharding enables horizontal scaling for write throughput.

Installation

Install the community server from mongodb.com.


MongoDB.live, free & fully virtual
mongoDB.
Cloud
Software
Learn
Solutions
Docs
Community Server
A free and open document database
Enterprise Server
Advanced features and security

The binaries are installed to the following location which we should add to our path.

C:\Program Files\MongoDB\Server\4.4\bin

I have chosen to install it into `C:\Users\rps\Code\temp\mongo` . I have also chosen custom and decided not to install it as a service as I am just playing around. Now we need to create a folder for data which I am going to call `C:\Users\rps\Code\temp\mongo\Data` Finally open a command prompt as an administrator and run the command

```
  mongod --dbpath C:\Users\rps\Code\temp\mongo\Data
```

By default, MongoDB listens on port 27017

The Shell

Assuming we have ran the server as per the previous section we can open the mongo shell. The shell is a complete JavaScript interpreter in addition to its mongo role. We can list the current database as follows.

```
db
>> test
```

To create a new database if it does not exist use the `use` command. If it does exist the use command will switch to that database. The database is not actually created until you first create a collection and insert a value into it. Before we enter the following command there is no database called `mydatabase` and no collection called `acollect`.

```
use mydatabase
```

Interview

How should we optimise aggregation pipeline?

Try and put matches at the start so they can take advantage of indices.