



Aggregation Framework in MongoDB



Aggregation framework

- Aggregation framework is a powerful tool offered by mongodb
- Using Aggregation framework you can easily group the documents based on some conditions and process it.
- You will get results pretty quickly
- Process the documents in stages



What is aggregation ?

- Aggregation basically groups the data from multiple documents and operates in many ways on those grouped data in order to return one combined result.
- In sql `count(*)` and with `group by` is an equivalent of MongoDB aggregation.
- Aggregate function groups the records in a collection.
- Can be used to provide total number(sum), average, minimum, maximum etc out of the group selected.



Three ways of performing aggregation

- Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.
- MongoDB provides three ways to perform aggregation:
 - [Aggregation pipeline](#)
 - [Map-reduce function](#)
 - [Single purpose aggregation methods](#).



Syntax for using Aggregate function

- In order to perform the aggregate function in MongoDB, aggregate () is the function to be used.
- Following is the syntax for aggregation :

```
db.collectionName.aggregate(pipeline options)
```

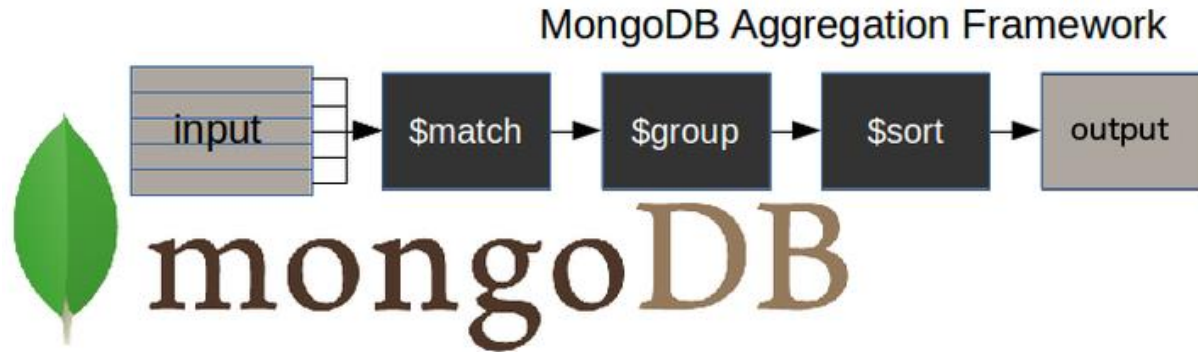


What is Pipeline?

- **Pipeline** : Aggregation Pipeline is a framework which performs aggregation for us.
- When we use Aggregation Framework, MongoDB pass document of a collection through a pipeline.
- In this pipeline document passes through different stages.
- Each stage change or transform the document and finally we get the computed result.

The MongoDB Aggregation Framework

- Here is a diagram to illustrate a typical pipeline.

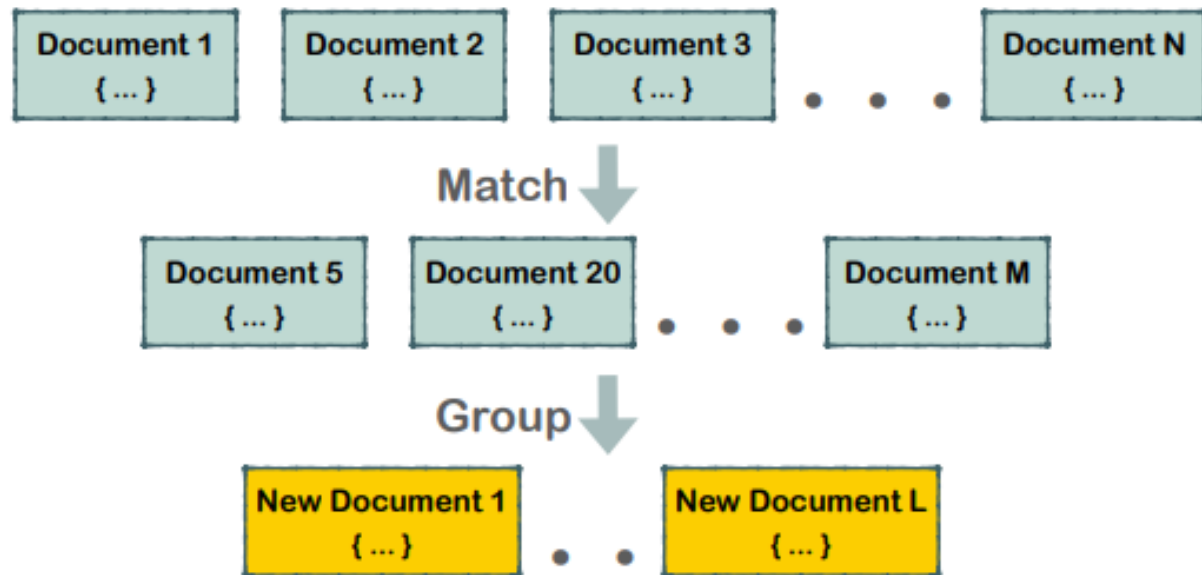


A decorative graphic on the left side of the slide. It features two fidget spinners, one red and one yellow, partially visible. A thick red curved line starts from the bottom left and extends towards the top right, passing behind the text.

What happens in pipeline

- The `$match()` stage filters those documents we need to work with, those that fit our needs.
- The `$group()` stage does the aggregation job and finally, we `$sort()` the resulting documents the way we require.
- In order to store the documents obtained we use the `$output()` stage.

Aggregation Process



A decorative graphic on the left side of the slide. It features two fidget spinners: one is red and the other is yellow. A thick red curved line starts from the bottom left and extends towards the top right, passing behind the text.

Aggregation process

- The input of the pipeline can be one or several collections.
- The pipeline then performs successive transformations on the data until our goal is achieved.
- This way, we can break down a complex query into easier stages, in each of which we complete a different operation on the data.
- So, by the end of the query pipeline, we will have achieved all that we wanted.
- This approach allows us to check whether our query is functioning properly at every stage by examining both its input and the output.
- The output of each stage will be the input of the next.



Example

- Group by age
- Group similar eye color persons
- Count the number of persons in USA
- Count the male population in USA



aggregate()

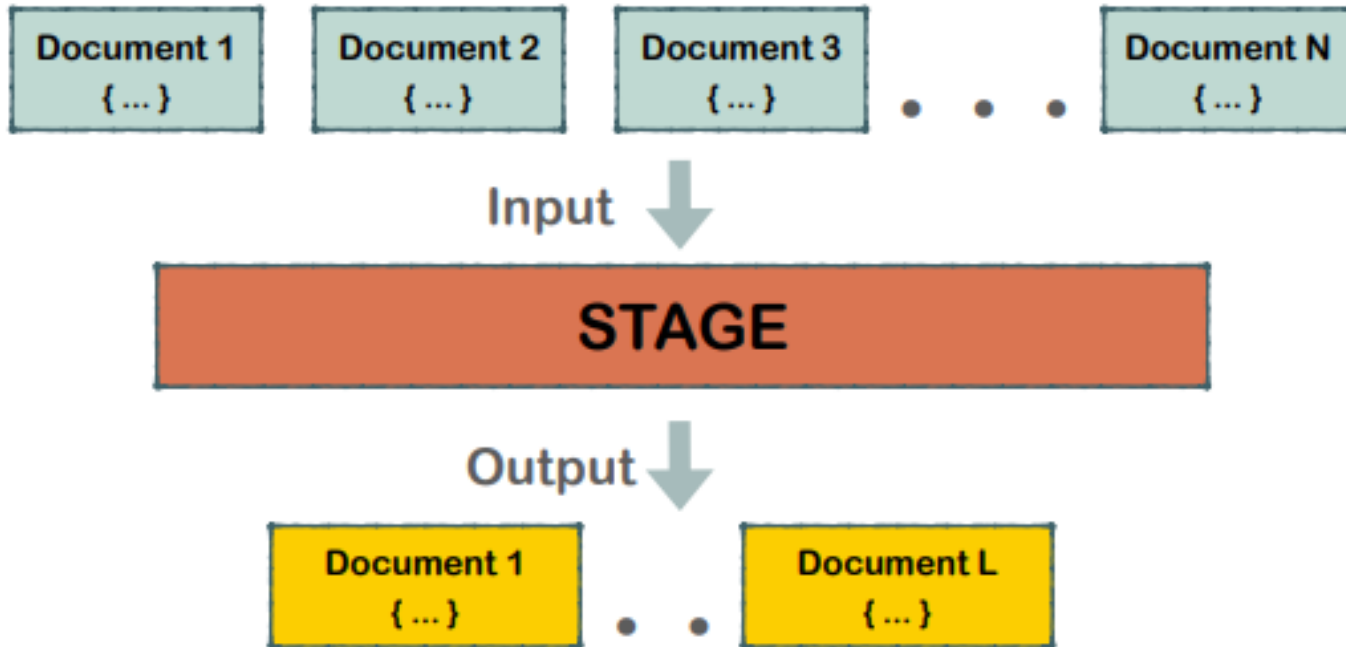
- Documents during aggregation process pass through the stages

```
db.<collection>.aggregate([  
    <stage1>,  
    <stage2>,  
    ...  
    <stageN>  
])
```

Note

Aggregation request
returns cursor from the
server

Aggregation Stage





Aggregation stage operators

- **\$project** – Used to select some specific fields from a collection.
- **\$match** – This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **\$group** – This does the actual aggregation as discussed above.
- **\$sort** – Sorts the documents.
- **\$skip** – With this, it is possible to skip forward in the list of documents for a given amount of documents.
- **\$limit** – This limits the amount of documents to look at, by the given number starting from the current positions.
- **\$unwind** – This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.



Aggregation in SQL and MongoDB

In SQL	In MongoDB	Description
Select	\$project	Passes the fields to next stage with existing Fields or with New fields.We can add new Fields dynamically
Where	\$match	This will filter the documents and will pass only matching documents to next pipeline stage.
Limit	\$limit	limit the first x unmodified documents and pass them to next stage of pipeline. x is the number of the documents which will pass through the next stage of pipeline.
GroupBy	\$group	This will group the documents and pass them to the next stage of Aggregation pipeline.
OrderBy	\$sort	It will change the order of documents either in ascending or descending.
Sum	\$sum	To calculate the sum of all the numeric values.
Join	\$lookup	It will perform the left outer join with another collection in same database.



Aggregation Stage Operators

- Each stage starts from the stage operator

```
{ $<stageOperator> : {} }
```

- Examples

```
{ $match: { age: { $gt: 20 } } }
```

```
{ $group: { _id: "$age" } }
```

```
{ $sort: { count: -1 } }
```

Aggregation Expressions

- Expression refers to the name of the field in input documents

`"$<fieldName>"`

Pass this
expression as
string

- Examples

```
{ $group: { _id: "$age" } }
```

```
{ $group: { _id: "$company.location.country" } }
```

```
{ $group: { _id: "$name", total: { $sum:  
"$price" } } }
```

\$match Stage

- Match specific documents using query

```
{ $match: { <query> } }
```

- Examples

```
{ $match: { city: "New York" } }
```

```
{ $match: {age: {$gt: 25} } }
```

```
{ $match: {$and: [ {gender: "female"} ,  
{age: {$gt: 25}} ] } }
```

Note

Match uses standard MongoDB queries and supports all query operators



\$match stage

- **\$match** is similar to **Where** in SQL. In SQL we use Where to filter the data and same is here.
- If we need to pass only a subset of our data in next stage of Aggregation Pipeline then we use **\$match**.
- **\$match** filters the data and pass the matching data to the next stage of Pipeline.



\$group stage

- MongoDB use **\$group** to group the documents by some specified expression.
- \$group is similar to Group clause in SQL.
- Group in SQL is not possible without any Aggregate Function and the same is here.
- We can not group in MongoDB without Aggregate Functions.

\$group Stage

- Groups input documents by certain expressions

```
{ $group: { _id: <expression>, <field1>:  
{ <accumulator1> : <expression1> }, ... } }
```

Note

_id is Mandatory field

- Examples

```
{ $group: { _id: "$age" } }  
{ $group: { _id: {age: "$age", gender:  
"$gender"} } }
```

_id is mandatory.
In _id we pass the field
on which we want to
group the documents



Example: 1

```
db.persons.aggregate(  
  [  
    { $group:{_id:{eyeColor:"$eyeColor"  
    }}  
    }  
  ]  
)
```

Example 2: \$group

```
db.persons.aggregate([  
  { $group: { _id: "$age" } }  
])
```



```
{ "_id" : 27 }  
{ "_id" : 30 }  
{ "_id" : 26 }  
{ "_id" : 31 }  
{ "_id" : 23 }  
{ "_id" : 37 }  
...
```



Example 3: group stage with multiple fields

```
db.persons.aggregate(  
  [  
    { $group:{_id:{eyeColor:"$eyeColor",  
      favoriteFruit:"$favoriteFruit"}}  
    }  
  ]  
)
```



Renaming field

```
db.persons.aggregate(  
  [  
    { $group:{_id:{age:"$age"  
    ,sex:"$gender"}}  
    }  
  ]  
)
```



Example 4: \$match and \$group stage

```
db.persons.aggregate([  
  { $match: { gender: "male" } },  
  { $group: { _id: { age: "$age", eyeColor: "$eyeColor" } } }  
])
```



```
{ "_id" : { "age" : 27, "eyeColor" : "blue" } }  
{ "_id" : { "age" : 21, "eyeColor" : "brown" } }  
{ "_id" : { "age" : 38, "eyeColor" : "brown" } }  
{ "_id" : { "age" : 25, "eyeColor" : "blue" } }  
{ "_id" : { "age" : 39, "eyeColor" : "blue" } }  
{ "_id" : { "age" : 27, "eyeColor" : "green" } }
```

...



Example 5: Swap the stage

\$group and \$match

```
db.persons.aggregate([
```

```
{
```

```
  $group: { _id: {age: "$age", eyeColor: "$eyeColor"} } },
```

```
  {$match:{gender:"male"}}
```

```
])
```

↓
EMPTY

Why this is
empty set

Wrong stages order



How to correct the Example 5

```
db.persons.aggregate([  
  { $group: { _id: {age: "$age", eyeColor:  
"$eyeColor"} } },  
  {$match:{"_id.eyeColor":"blue"}}  
  
])
```

Example 6: Another example on \$group and \$match

```
db.persons.aggregate([  
  { $group: { _id: {age: "$age", eyeColor: "$eyeColor"} } },  
  { $match: { "_id.age": {$gt: 30} } }  
])
```



```
{ "_id" : { "age" : 38, "eyeColor" : "brown" } }  
{ "_id" : { "age" : 33, "eyeColor" : "green" } }  
{ "_id" : { "age" : 35, "eyeColor" : "brown" } }  
{ "_id" : { "age" : 37, "eyeColor" : "green" } }  
{ "_id" : { "age" : 39, "eyeColor" : "brown" } }  
...
```



\$count stage

- Returns a document that contains a count of the number of documents input to the stage.
- \$count has the following prototype form:
- **{ \$count: <string> }**
- <string> is the name of the output field which has the count as its value. <string> must be a non-empty string, must not start with \$ and must not contain the . character.



\$count stage

- Counts number of the input documents

```
{ $count: "<title>" }
```

- Example

```
{ $count: "countries" }
```



```
{ "countries" : 4 }
```

A decorative graphic on the left side of the slide. It features two fidget spinners: one is red and the other is yellow. A thick red arc curves from the bottom left towards the top right, passing behind the text.

Example 1 on \$count stage

```
db.persons.aggregate([  
  { $count: "allDocumentsCount"}  
])
```




Example : \$group and \$count

```
db.persons.aggregate([  
  { $group: { _id: "$eyeColor" } },  
  { $count: " eyeColor "  
})
```

```
personData 0.001 sec.  
1 /* 1 */  
2 {  
3   " eyeColor " : 3  
4 }
```



Example 3: \$group and \$count

```
db.persons.aggregate([  
  { $group: { _id: {eyeColor:"$eyeColor"  
    ,gender:"$gender"} } },  
  { $count: " eyeColor AndGender"  
  }  
])
```



Example: match,group,count

```
db.persons.aggregate([  
  {$match:{age:{$gte:25}}},  
  {$group:{_id:{eyeColor:"$eyeColor",  
    age:"$age" }}},  
  //{$count:"EyeColorAndAge"}  
])
```

\$sort Stage

- Sorts input documents by certain field(s)

```
{ $sort: { <field1>: <-1 | 1>, <field2>: <-1 | 1> ... } }
```

- Examples

```
{ $sort: {score: -1} }
```

```
{ $sort: {age: 1 , country: 1} }
```

Note

<field>: 1 Ascending Order

<field>: -1 Descending Order

A decorative graphic on the left side of the slide. It features two fidget spinners: one is red and the other is yellow. A thick red curved line starts from the bottom left and arcs upwards towards the right, passing behind the text.

Example: sort

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



Example: group and sort

```
db.persons.aggregate([  
  { $group: { _id: "$favoriteFruit" } },  
  { $sort: { _id: 1 } }  
])
```



```
{ "_id" : "apple" }  
{ "_id" : "banana" }  
{ "_id" : "strawberry" }
```



Example : with \$group and \$match

```
db.personData.aggregate([  
  {$group:{_id: {eyeColor:"$eyeColor"  
    ,favoriteFruit:"$favoriteFruit"  
    }}}  
] )
```




\$group and \$sort

```
db.persons.aggregate([  
  {$group:{_id: {eyeColor:"$eyeColor"  
    ,favoriteFruit:"$favoriteFruit"  
    }},  
  {$sort:{"_id.eyeColor":1,"_id.favoriteFruit":-1}}  
] )
```



\$match,\$group and \$sort

```
db.persons.aggregate([  
  {$match:{eyeColor:{$ne:"blue"}}  
  },  
  {$group:{_id: {eyeColor:"$eyeColor"  
    ,favoriteFruit:"$favoriteFruit"  
    }  
  }},  
  {$sort:{"_id.eyeColor":1,"_id.favoriteFruit":-  
1}}  
] )
```



\$project

- We can select certain fields, rename Fields from documents though **\$project**.
- In short **\$project** reshape the documents by adding/removing or renaming the documents for the next stage of pipeline.
- In \$project we use 1 or true if we want to include the Field and 0 or false if we want to exclude a particular field.
- If we have passed n input documents, we will have n output documents
- \$project stage immediately goes after the \$match stage



\$project example

```
db.persons.aggregate([  
  {$project:{name:1,age:1}}])
```

Description:

Retrieves name and age field from the input document and passes to the next stage



What would be the output of the following?

```
db.personData. aggregate([  
  {$project:{isActive:0,gender :0,eyeColor:0}}])
```



What would be the output of the following?

```
db.personData. aggregate([  
  {$project:{isActive:0,gender :0,eyeColor:0}}])
```

Outputs all the field except the specified fields



\$limit stage

\$limit operator use to pass n documents to next pipe line stage where n is the limit.n is the number of documents

- **Outputs first N documents from the input**

```
{ $limit: <number> }
```

- **Examples**

```
{ $limit: 100 }
```

```
{ $limit: 1000 }
```

Note

\$limit is usually used in:

1. Sampled aggregation requests with \$limit as first stage
2. After \$sort to produce topN results



Example : \$limit, \$match and \$group

```
db.personData.aggregate([  
  { $limit: 5},  
  { $match: { age: { $gt: 25}} },  
  { $group: { _id: "$name" } }  
])
```



\$unwind stage

- Splits each document with specified array to several documents - one document per array element

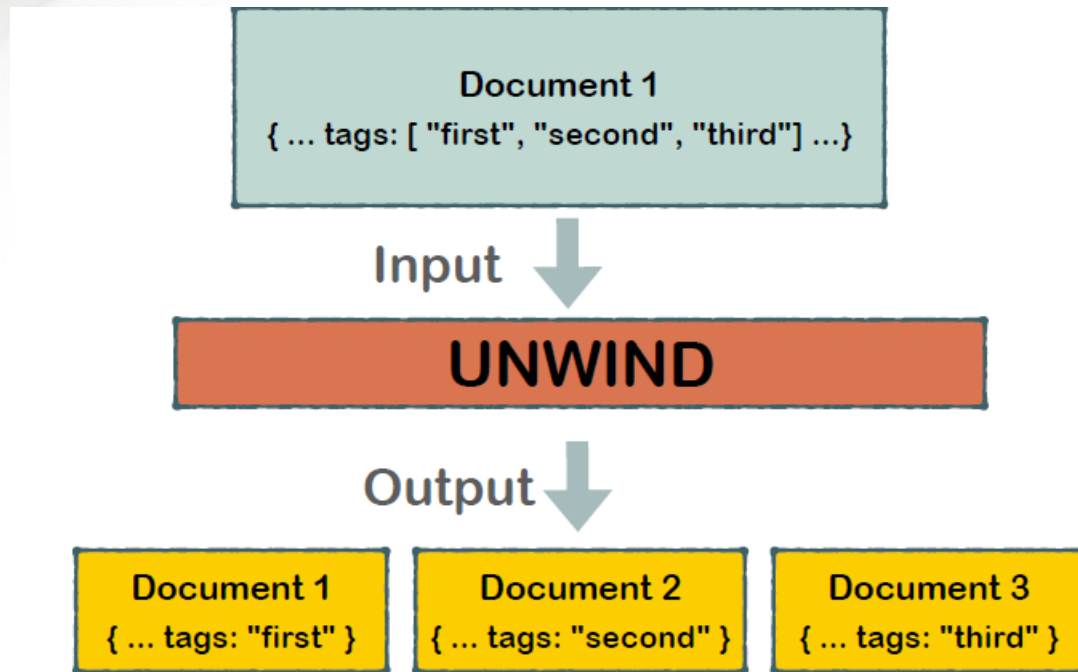
```
{ $unwind: <arrayReferenceExpression> }
```

- Examples

```
{ $unwind: "$tags" }
```

```
{ $unwind: "$hobbies" }
```

\$unwind stage logic



A decorative graphic on the left side of the slide. It features two fidget spinners: one is red and the other is yellow. A thick red curved line starts from the bottom left and sweeps upwards towards the right, passing behind the text.

\$unwind example

```
$unwind  
db.PersonArray.aggregate(  
[  
  {$unwind:"$tags"} ] )
```



Accumulators

\$sum


\$avg

\$max

\$min

Note

Most accumulators are used only
in the \$group stage





Accumulators Syntax

- Accumulators maintain state for each group of the documents

```
{ $<accumulatorOperator>: <expression> }
```

- Examples

```
{ $sum: "$quantity" }
```

```
{ $avg: "$age" }
```


\$sum Accumulator

- Sums numeric values for the documents in each group

```
{ $sum: <expression | number> }
```

- Examples

```
{ total: { $sum: "$quantity" } }  
{ count: { $sum: 1 } }
```



Simple way to count number of the documents in each group



Example: \$sum and \$group

```
db.personData.aggregate([  
  {  
    $group: {  
      _id: "$age",  
      count: { $sum: 1 }  
    }  
  }  
])
```



Example: \$sum and \$group

```
db.personData.aggregate([
  {
    $group: {
      _id: "$age",
      count: { $sum: 1 }
    }
  }
])
```

{ "_id" : 27, "count" : 42 }
{ "_id" : 30, "count" : 38 }
{ "_id" : 26, "count" : 51 }
{ "_id" : 31, "count" : 53 }
{ "_id" : 23, "count" : 57 }
{ "_id" : 37, "count" : 49 }
{ "_id" : 32, "count" : 38 }
...



\$avg Accumulator

- Calculates average value of the certain values in the documents for each group

```
{ $avg: <expression> }
```

- Example

```
{ avgAge: { $avg: "$age" } }
```



Example: \$sum and \$avg

```
db.personData.aggregate([  
  {  
    $group: {  
      _id: "$eyeColor",  
      avgAge: { $avg: "$age" }  
    }  
  }  
])
```



Example: \$sum and \$avg

```
db.personData.aggregate([  
  {  
    $group: {  
      _id: "$eyeColor",  
      avgAge: { $avg: "$age" }  
    }  
  }  
])
```

```
{ "_id" : "brown", "avgAge" : 29.816023738872403 }  
{ "_id" : "blue", "avgAge" : 30.033033033033032 }  
{ "_id" : "green", "avgAge" : 29.654545454545456 }  
...
```

Unary Operators

\$type

\$or

\$!t

\$gt

\$and

\$multiply

Note

1. Unary Operators are usually used in the \$project stage
2. In the \$group stage Unary Operators can be used only in conjunction with Accumulators



\$type Unary Operator

- Returns BSON type of the field's value

```
{ $type: <expression> }
```

- Examples

```
{ $type: "$age" }
```

```
{ $type: "$name" }
```




Example : \$type and \$project

```
db.personData.aggregate([  
  {  
    $project: {  
      name: 1,  
      eyeColorType: { $type: "$eyeColor" },  
      ageType: { $type: "$age" }  
    }  
  }  
])
```



Example : \$type and \$project

```
db.personData.aggregate([
{
  $project: {
    name: 1,
    eyeColorType: { $type: "$eyeColor" },
    ageType: { $type: "$age" }
  }
}]
{
  "_id" : ObjectId("5ad4cbde2edbf6ddeec71741"),
  "name" : "Aurelia Gonzales",
  "eyeColorType" : "string",
  "ageType" : "int"
}
```



\$out Stage

- Writes resulting documents to the MongoDB collection

```
{ $out: "<outputCollectionName>" }
```

- Example

```
{ $out: "newCollection" }
```

Notes

- \$out MUST be last stage in the pipeline
- If output collection doesn't exist, it will be created automatically



Example : \$out

```
db.personData.aggregate([  
  { $group: { _id: {age: "$age", eyeColor:  
    "$eyeColor"} }},  
  { $out: "aggregationResults"  
  }  
])
```



Documents from the \$group
stage will be written to
the collection
"aggregationResults"

allowDiskUse: true

- ◎ All aggregation stages can use maximum 100 MB of RAM
- ◎ Server will return error if RAM limit is exceeded
- ◎ Following option will enable MongoDB to write stages data to the temporal files

```
{ allowDiskUse: true }
```

◎ Example

```
db.persons.aggregate([], {allowDiskUse: true})
```

SUMMARY

- **Aggregation Stages**

 - \$group

 - \$match

 - \$sort

 - \$project

 - \$out

- **Stages chaining**

- **Accumulator Operators**

 - \$sum

 - \$avg

- **Unary Operators**