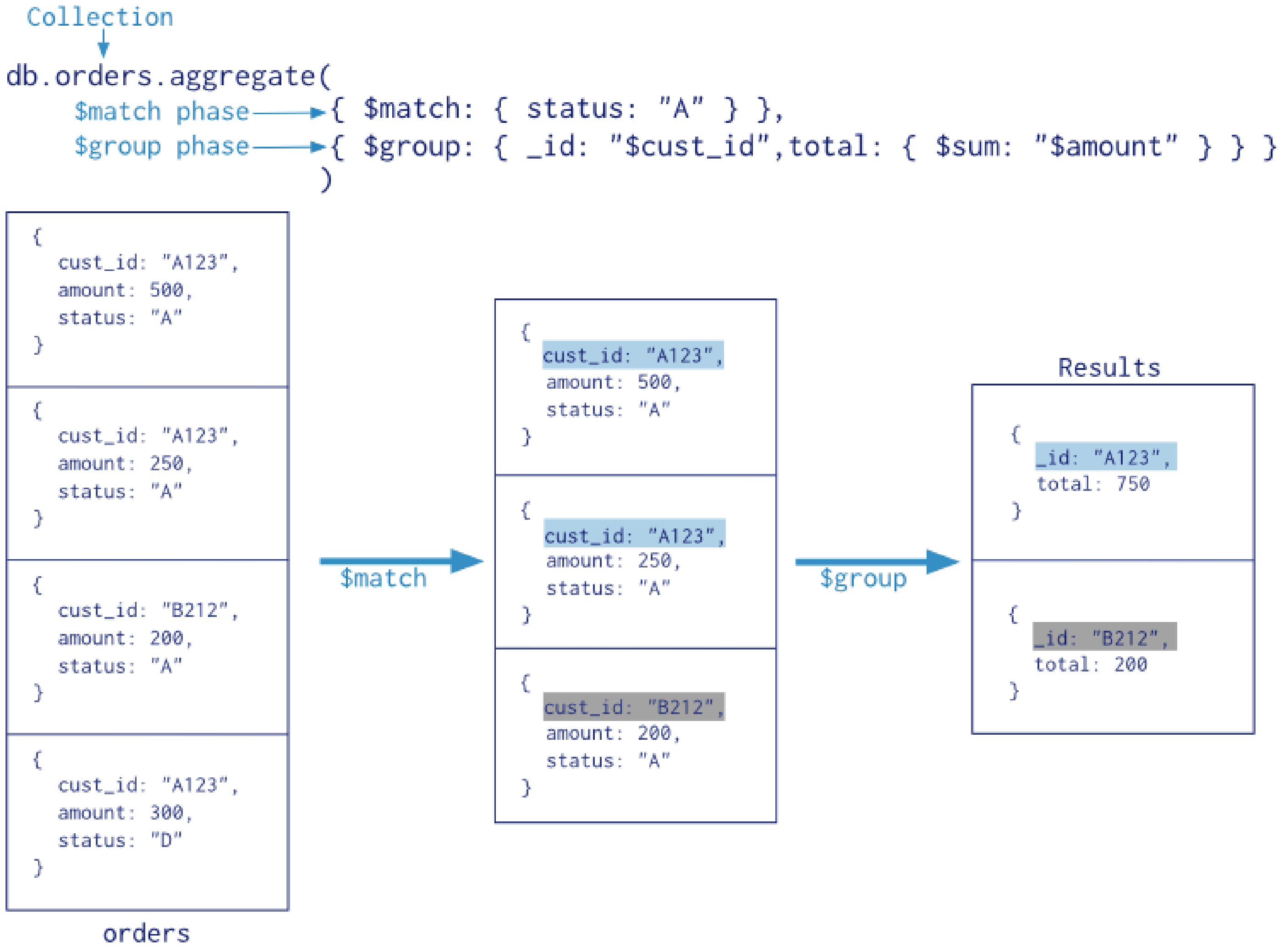
MONGODB AGGREGATION PIPELINE

https://docs.mongodb.com/manual/core/aggregation-pipeline/

Rev.2

AT A GLANCE



```
db.orders.aggregate(
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```

orders

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
{
   cust_id: "A123",
   amount: 300,
   status: "D"
}
```

```
{
   cust_id: "A123",
   amount: 500,
   status: "A"
}
{
   cust_id: "A123",
   amount: 250,
   status: "A"
}
{
   cust_id: "B212",
   amount: 200,
   status: "A"
}
```

Results

```
{
   _id: "A123",
   total: 750
}
{
   _id: "B212",
   total: 200
}
```

GENERAL CONCEPTS

- Documents enter a multi-stage pipeline that transforms the **documents of a collection** into an aggregated result
- Pipeline **stages** can appear **multiple** times in the pipeline
  - exceptions *$out, $merge,* and *$geoNear* stages
- Pipeline expressions can **only** operate on the **current document** in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents (max 100 Mb of RAM per stage).
- Generally, expressions are **stateless** and are only evaluated when seen by the aggregation process with one exception: accumulator expressions used in the *$group* stage (e.g. totals, maximums, minimums, and related data).
- The aggregation pipeline provides an alternative to ***map-reduce*** and may be the preferred solution for aggregation tasks since MongoDB introduced the *$accumulator* and *$function* aggregation operators starting in version 4.4

COMPARISON WITH SQL

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM	$sum
COUNT	$sum

PIPELINE

- Aggregate functions can be applied to collections to group documents

```
db.collection.aggregate( [ <list of stages> ] )
```

- Common stages: `$match, $group` ..
- The aggregate function allows applying aggregating functions (e.g. sum, average)
- It can be combined with an initial definition of groups based on the grouping fields

EXAMPLE

```
db.people.aggregate( [
  { $group: { _id: null,
              mytotal: { $sum: "$age" },
              mycount: { $sum: 1 }
            }
  }
] )
```

- Considers all documents of people and
  - sum the values of their age
  - sum a set of ones (one for each document)
- The returned value is associated with a field called “mytotal” and a field “mycount”

EXAMPLE

```
db.people.aggregate( [
  { $group: { _id: null,
              myaverage: { $avg: "$age" },
              mytotal: { $sum: "$age" }
             }
  }
] )
```

- Considers all documents of people and computes
  - sum of age
  - average of age

EXAMPLE

```
db.people.aggregate( [
  { $match: {status: "A"} } ,
  { $group: { _id: null,
                   count: { $sum: 1 }
                  }
  }
] )
```

- Counts the number of documents in people with status equal to “A”

EXAMPLE

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

- Creates one group of documents for each value of status and counts the number of documents per group
  - returns one value for each group containing the value of the grouping field and an integer representing the number of documents

GROUP BY

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
  {
    $group: {
      _id: "$status",
      total: { $sum: "$age" }
    }
  }
] )
```

Group field

Aggregation function

GROUP BY & HAVING

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
      SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
  {
    $group: {
      _id: "$status",
      total: { $sum: "$age" }
    }
  },
 { $match: { total: { $gt: 1000 }
} }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in HAVING

PIPELINE STAGES

Stage	Description
$addFields	Adds **new fields** to documents. Reshapes each document by adding new fields to output documents that will contain both the existing fields from the input documents and the newly added fields.
$bucket	Categorizes incoming documents **into groups**, called buckets, based on a specified expression and bucket boundaries. On the contrary, **$group** creates a "bucket" for each value of the group field.
$bucketAuto	Categorizes incoming documents into a specific number of groups, called buckets, based on a specified expression. Bucket boundaries are automatically determined in an attempt to **evenly distribute** the documents into the specified number of buckets.
$collStats	Returns statistics regarding a collection or view (it must be the **first stage**)
$count	Passes a document to the next stage that contains a count of the input **number of documents** to the stage (same as $group+$project)

PIPELINE STAGES

Stage	Description
$facet	Processes **multiple aggregation pipelines** within a single stage on the same set of input documents. Enables the creation of multi-faceted aggregations capable of characterizing data across multiple dimensions. Input documents are passed to the $facet stage only once, without needing multiple retrieval.
$geoNear	Returns an ordered stream of documents based on the **proximity** to a geospatial point. The output documents include an additional **distance** field. It must in the **first stage** only.
$graphLookup	Performs a **recursive search** on a collection. To each output document, adds a new array field that contains the traversal results of the recursive search for that document.

EXAMPLE

```
db.employees.aggregate( [
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
] )
```

- The $graphLookup operation recursively matches on the **reportsTo** and **name** fields in the employees collection, returning the **reporting hierarchy** for each person.
- Returns a list of documents such as

```
{
  "_id" : 5,
  "name" : "Asya",
  "reportsTo" : "Ron",
  "reportingHierarchy" : [
    { "_id" : 1, "name" : "Dev" },
    { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
    { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
  ]
}
```

original document

PIPELINE STAGES

Stage	Description
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$indexStats	Returns statistics regarding the use of each index for the collection.
$limit	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$lookup	Performs a **join** to another collection in the *same* database to filter in documents from the "joined" collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the "joined" collection. The $lookup stage passes these reshaped documents to the next stage.

PIPELINE STAGES

Stage	Description
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$merge	Writes the resulting documents of the aggregation pipeline to a collection. The stage can incorporate (insert new documents, merge documents, replace documents, keep existing documents, fail the operation, process documents with a custom update pipeline) the results into an output collection. To use the $merge stage, it must be the last stage in the pipeline.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

PIPELINE STAGES

Stage	Description
$sample	Randomly selects the specified number of documents from its input.
$set	Adds new fields to documents. Similar to $project, $set reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields. $set is an alias for **$addFields** stage. If the name of the new field is the same as an existing field name (including _id), $set **overwrites** the existing value of that field with the value of the specified expression.
$skip	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.

PIPELINE STAGES

Stage	Description
$sortByCount	Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
$unset	Removes/excludes fields from documents.
$unwind	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.

18

EXAMPLES

DATA MODEL

Given the following collection of books

```
{_id:ObjectId("5fb29ae15b99900c3fa24292"),
 title:”MongoDb Guide",
 tag:[”mongodb”,”guide”,“database”],
 n:100,
 review_score:4.3,
 price:[{v: 19.99, c: “€”,  country: “IT”},
        {v: 18, c: “£”, country:”UK”} ],
 author: {_id: 1,
          name:”Mario”,
          surname: “Rossi”}
},
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

EXAMPLE 1

For each **country**, select the average **price** and the average **review_score**.

The review score should be rounded down.

Show the first 20 results with a total number of books higher than 50.

$UNWIND

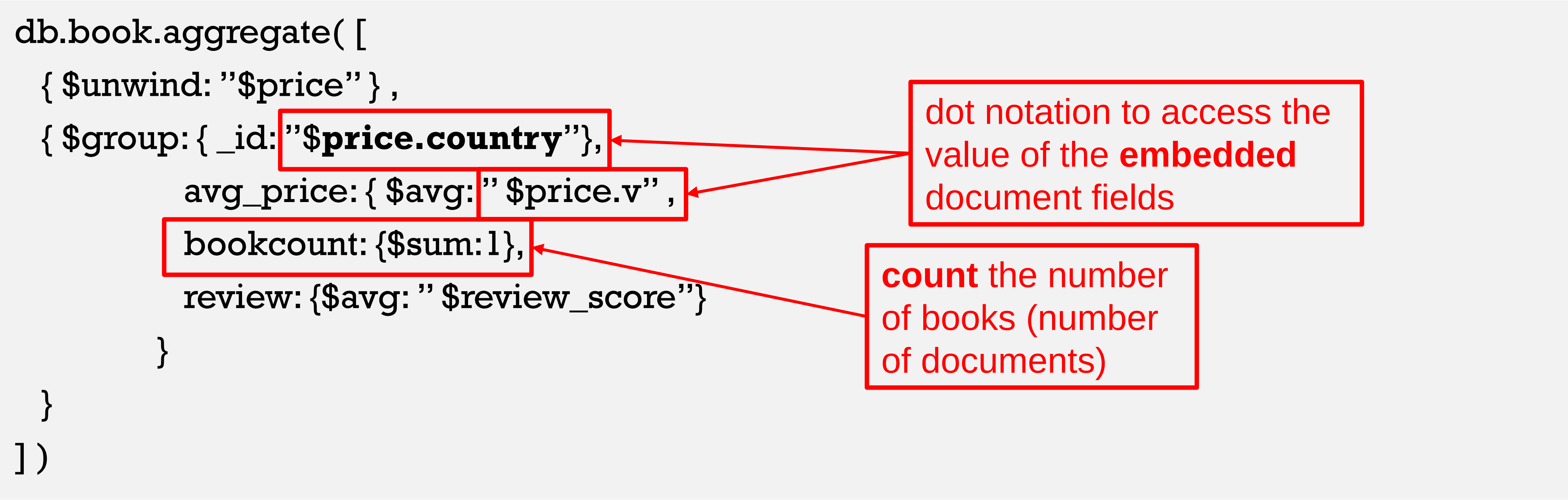
```
db.book.aggregate( [
  { $unwind: ”$price” } ,
] )
```

Build a document for each entry of the **price** array

RESULT - $UNWIND

```
{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : [ "mongodb",
"guide", "database" ], "n" : 100, "review_score" : 4.3, "price" : { "v" : 19.99, "c" : " € ", "country" : "IT" },
"author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : [ "mongodb",
"guide", "database" ], "n" : 100, "review_score" : 4.3, "price" : { "v" : 18, "c" : "£", "country" : "UK" },
"author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : [
"python", "guide", "programming" ], "n" : 352, "review_score" : 4.6, "price" : { "v" : 24.99, "c" : " € ",
"country" : "IT" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : [
"python", "guide", "programming" ], "n" : 352, "review_score" : 4.6, "price" : { "v" : 19.49, "c" : "£",
"country" : "UK" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

…
```

$GROUP



```
db.book.aggregate( [
  { $unwind: ”$price” } ,
  { $group: { _id: ”$price.country”},
              avg_price: { $avg: ” $price.v” ,
              bookcount: {$sum: 1},
              review: {$avg: ” $review_score”}
            }
  }
] )
```

dot notation to access the value of the **embedded** document fields

count the number of books (number of documents)

RESULT - $GROUP

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}
{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}
{ "_id" : "US", "avg_price" : 22.49, "bookcount": 49, "review": 4.2}
…
```

$MATCH

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
              avg_price: { $avg: '$price.v' },
              bookcount: {$sum:1},
              review: {$avg: '$review_score'}
            }
  },
 {$match: { bookcount: { $gte: 50 } }},
] )
```

Filter the documents where **bookcount** is greater than 50

RESULT - $GROUP

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}

{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}

…
```

$PROJECT

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
] )
```

round down the review score

RESULT - $PROJECT

```
{ "_id" : "UK", "avg_price" : 18.75, "review": 4}
{ "_id" : "IT", "avg_price" : 22.49, "review" :  3}
…
```

$LIMIT

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
             }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
{$limit:20}
] )
```

Limit the results to the first 20 documents

EXAMPLE 2

Compute the **95 percentile** of the **number of pages,**
only for the books that contain the **tag** “guide”.

$MATCH

```
db.book.aggregate( [
   {$match:{ tag : "guide"} }
] )
```

select documents containing “guide” in the tag array, compare with tag:[“guide”]

RESULT - $MATCH

```
{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : [
"python", "guide", "programming" ], "n" : 352, "review_score" : 4.6, "price" : [ { "v" : 24.99, "c" :
"€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" } ], "author" : { "_id" : 1, "name" :
"John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : [
"mongodb", "guide", "database" ], "n" : 100, "review_score" : 4.3, "price" : [ { "v" : 19.99, "c" :
"€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" } ], "author" : { "_id" : 1, "name" :
"Mario", "surname" : "Rossi" } }


…
```

$SORT

```
db.book.aggregate( [
  {$match: { tag : "guide"} },
 {$sort : { n: 1} }
] )
```

sort the documents in ascending order according to the value of the **n** field, which stores the number of pages of each book

RESULT - $SORT

```
{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : [
"mongodb", "guide", "database" ], "n" : 100, "review_score" : 4.3, "price" : [ { "v" : 19.99, "c" :
"€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" } ], "author" : { "_id" : 1, "name" :
"Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : [
"python", "guide", "programming" ], "n" : 352, "review_score" : 4.6, "price" : [ { "v" : 24.99, "c" :
"€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" } ], "author" : { "_id" : 1, "name" :
"John", "surname" : "Black" } }


…
```

$GROUP + $PUSH

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
 {$sort : { n: 1} },
 {$group: {_id:null, value: {$push: "$n"}}}
] )
```

group all the records together inside a single document (**_id:null**), which contains an array with all the values of **n** of all the records

RESULT - $GROUP + $PUSH

```
{ "_id": null, "value": [100, 352, ...]}
```

$PROJECT + $ARRAYELEMAT

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
 {$sort : { n: 1} },
 {$group: {_id:null, value: {$push: "$n"}}},
 {$project:
            {"n95p": {$arrayElemAt:
                      ["$value",
                      {$floor: {$multiply: [0.95, {$size: "$value"}]}}
                     ]
            } }
}
] )
```

get the value of the array at a given index with { $arrayElemAt: [<array>, <idx>] }

compute the index at 95% of the array length

RESULT - $PROJECT + $ARRAELEMAT

```
{ "_id" : null, "n95p" : 420 }
```

EXAMPLE 3

Compute the **median** of the **review_score,**

only for the books having at least a **price** whose **value** is higher than 20.0.

SOLUTION

```
db.book.aggregate( [
 {$match: {'price.v' : { $gt: 20 }} },
 {$sort : {review_score: 1} },
 {$group: {_id:null, rsList: {$push: '$review_score'}}},
 {$project:
            {'median': {$arrayElemAt:
                          ['$rsList',
                          {$floor: {$multiply: [0.5, {$size: '$rsList'}]}}
                   ]
            } }
}
] )
```