Large-Scale and Multi-Structured Databases

Projection operations and Aggregations in MongoDB

Prof Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Copyright Issues

Most of the information included this presentation have been extracted from the official documentation of MongoDB.

Check on the last slide the sources used to extract the material for this class.

Project Fields to Return from Query

By default, queries in MongoDB return all fields in matching documents.

To limit the amount of data that MongoDB sends to applications, we can include a ***projection document*** to specify or restrict ***fields*** to return.

Let consider to upload the following collection:

```
db.inventory.insertMany( [
  { item: "journal", status: "A", size: { h: 14, w: 21, uom: "cm" }, instock: [ { warehouse: "A", qty: 5 } ] },
  { item: "notebook", status: "A",  size: { h: 8.5, w: 11, uom: "in" }, instock: [ { warehouse: "C", qty: 5 } ] },
  { item: "paper", status: "D", size: { h: 8.5, w: 11, uom: "in" }, instock: [ { warehouse: "A", qty: 60 } ] },
  { item: "planner", status: "D", size: { h: 22.85, w: 30, uom: "cm" }, instock: [ { warehouse: "A", qty: 40 } ] },
  { item: "postcard", status: "A", size: { h: 10, w: 15.25, uom: "cm" }, instock: [ { warehouse: "B", qty: 15 }, {
warehouse: "C", qty: 35 } ] }
]);
```

Return the Specified Fields

A ***projection*** can explicitly include several fields by setting the ***<field> to 1*** in the projection document.

In the following example, the operation returns all documents that match the query, limited to fields ***item*** and ***status***.

```
> db.inventory.find( { status: "A" }, { item: 1, status: 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A" }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
{ "item" : "journal", "status" : "A" }
{ "item" : "notebook", "status" : "A" }
{ "item" : "postcard", "status" : "A" }
```

This object specifies to suppress the _id field.

Return All But the Excluded Fields

Instead of listing the fields to return in the matching document, you can use a projection ***to exclude*** specific fields.

The following example which returns all fields ***except for the status and the instock fields*** in the matching documents:

```
> db.inventory.find( { status: "A" }, { status: 0, instock: 0 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "size" : { "h" : 14, "w" : 21, "uom" : "cm" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "size" : { "h" : 8.5, "w" : 11, "uom" : "in" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" } }
```

With the exception of the _id field, we **cannot combine inclusion and exclusion** statements in projection documents.

Return Specific Fields in Embedded Documents

We can return ***specific fields*** in an ***embedded document***.

Use the dot notation to refer to the embedded field and ***set to 1*** in the projection document.

Check the following example:

```
> db.inventory.find( { status: "A" }, { item: 1, status: 1, size:1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "size" : { "h" : 14, "w" : 21, "uom" : "cm" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "size" : { "h" : 8.5, "w" : 11, "uom" : "in" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" } }
> db.inventory.find(    { status: "A" },    { item: 1, status: 1, "size.uom": 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "size" : { "uom" : "cm" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "size" : { "uom" : "in" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "size" : { "uom" : "cm" } }
```

Suppress Specific Fields in Embedded Documents

We can ***suppress specific fields*** in an embedded document.

Use the dot notation to refer to the embedded field in the projection document and ***set to 0***.

Check the following example:

```
> db.inventory.find( { status: "A" }, { status: 0, instock: 0, "size.uom": 0  } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "size" : { "h" : 14, "w" : 21 } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "size" : { "h" : 8.5, "w" : 11 } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "size" : { "h" : 10, "w" : 15.25 } }
> db.inventory.find( { status: "A" }, { status: 0, instock: 0, } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "size" : { "h" : 14, "w" : 21, "uom" : "cm" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "size" : { "h" : 8.5, "w" : 11, "uom" : "in" } }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "size" : { "h" : 10, "w" : 15.25, "uom" : "cm" } }
```

Projection on Embedded Documents in an Array

The following example specifies a projection to return:

- The *_id* field (returned by default),
- The *item* field,
- The *status* field,
- The *qty* field in the documents embedded in the *instock* array.

Check the following example:

```
> db.inventory.find( { status: "A" }, { item: 1, status: 1, "instock.qty": 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "instock" : [ { "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "instock" : [ { "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "instock" : [ { "qty" : 15 }, { "qty" : 35 } ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 }, { "warehouse" : "C", "qty" : 35 } ] }
```

Project Specific Array Elements in the Returned Array

For fields that contain arrays, MongoDB provides the following projection operators for manipulating arrays: ***$elemMatch***, ***$slice***, and ***$***.

Check the following example:

```
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: 1 } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 }, { "warehouse" : "C", "qty" : 35 } ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $slice: -1 } } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 35 } ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $slice: 1 } } )
{ "_id" : ObjectId("5db958068215ec9bc20c6c75"), "item" : "journal", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c76"), "item" : "notebook", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 5 } ] }
{ "_id" : ObjectId("5db958068215ec9bc20c6c79"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 } ] }
```

We cannot project specific array elements using the array index; e.g. { "instock.0": 1 }

$ElemMatch Usage

The ***$elemMatch*** operator project the first matching element from an array based on a condition.

```
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: 1} )
{ "_id" : ObjectId("5fb290800ad2fdf2c5553832"), "item" : "journal", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 5 } ] }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553833"), "item" : "notebook", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 5 } ] }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553836"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 }, { "warehouse" : "C", "qty" : 35 } ] }
{ "_id" : ObjectId("5fb290aa0ad2fdf2c5553837"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 }, { "warehouse" : "C", "qty" : 35 }, { "warehouse" : "A", "qty" : 5 }, { "warehouse" : "A", "qty" : 60 }
 ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $elemMatch: { qty: { $gt: 20 } } } } )
{ "_id" : ObjectId("5fb290800ad2fdf2c5553832"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553833"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553836"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 35 } ] }
{ "_id" : ObjectId("5fb290aa0ad2fdf2c5553837"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "C", "qty" : 35 } ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $elemMatch: { qty: { $gt: 10 } } } } )
{ "_id" : ObjectId("5fb290800ad2fdf2c5553832"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553833"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553836"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 } ] }
{ "_id" : ObjectId("5fb290aa0ad2fdf2c5553837"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "B", "qty" : 15 } ] }
> db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $elemMatch: { qty: { $gt: 50 } } } } )
{ "_id" : ObjectId("5fb290800ad2fdf2c5553832"), "item" : "journal", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553833"), "item" : "notebook", "status" : "A" }
{ "_id" : ObjectId("5fb290800ad2fdf2c5553836"), "item" : "postcard", "status" : "A" }
{ "_id" : ObjectId("5fb290aa0ad2fdf2c5553837"), "item" : "postcard", "status" : "A", "instock" : [ { "warehouse" : "A", "qty" : 60 } ] }
>
```

Query for Null Fields

Let consider the following simple collection:

db.inventory.insertMany([
{ _id: 1, item: null },
{ _id: 2 },
{ _id: 3, item: “book” }])

The ***{ item : null }*** query matches documents that either contain the item field whose value is null or that do not contain the item field. Check the following example:

```
> db.inventory.find();
{ "_id" : 1, "item" : null }
{ "_id" : 2 }
{ "_id" : 3, "item" : "book" }
> db.inventory.find( { item: null } )
{ "_id" : 1, "item" : null }
{ "_id" : 2 }
```

Type Check

It is possible to query documents that contains field of a ***specific type***.

Check the following example:

```
> db.inventory.find( { item : { $type: 2 } } )
{ "_id" : 3, "item" : "book" }
> db.inventory.find( { item : { $type: 10 } } )
{ "_id" : 1, "item" : null }
```

https://www.mongodb.com/docs/manual/reference/operator/query/type/#mongodb-query-op.-type

Query for Missing Fields

The following examples query for documents that contain/do not contain a specific field:

```
> db.inventory.find( { item : { $exists: true } } )
{ "_id" : 1, "item" : null }
{ "_id" : 3, "item" : "book" }
> db.inventory.find( { item : { $exists: false } } )
{ "_id" : 2 }
```

Aggregation

Introduction

Aggregation operations ***process data records*** and ***return computed results***.

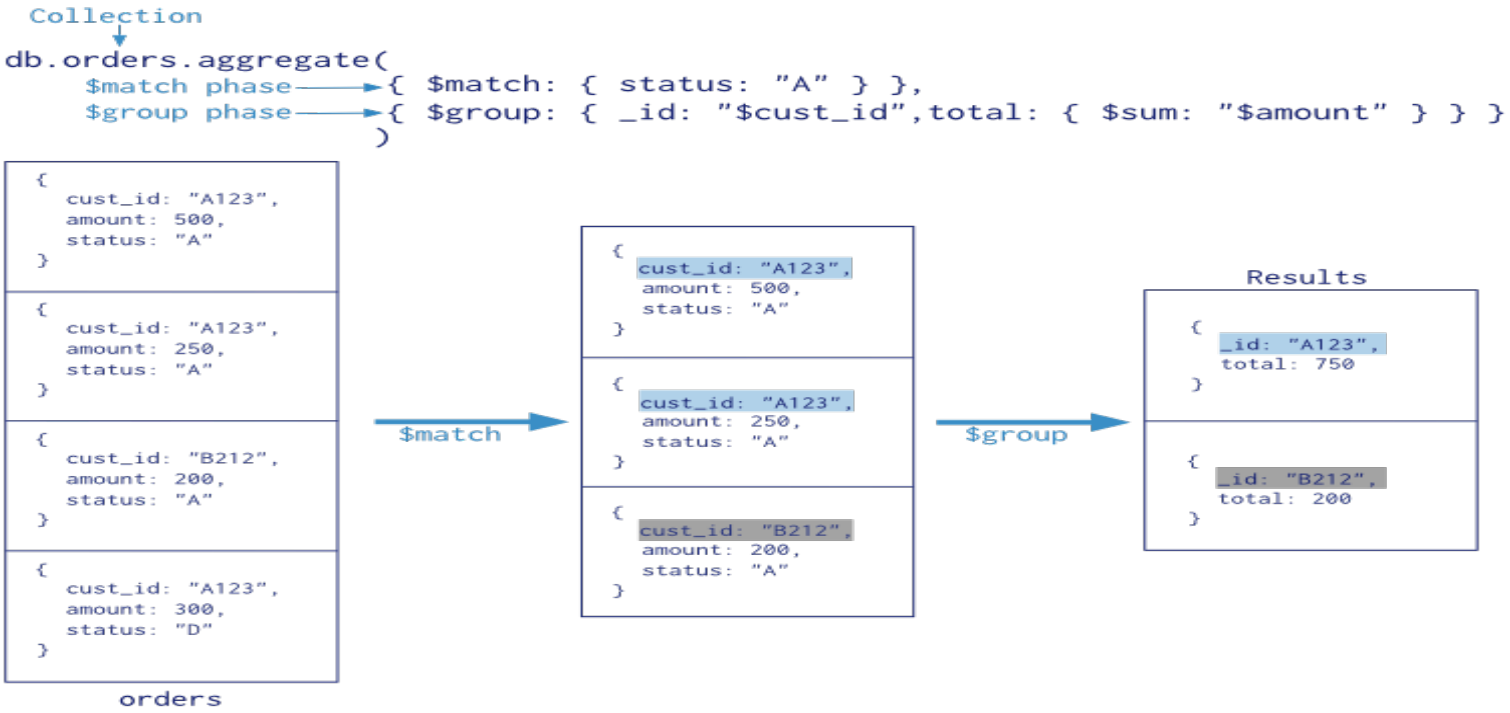
Aggregation operations ***group*** values from multiple documents together and can perform a ***variety of operations*** on the grouped data to ***return a single result***.

MongoDB provides three ways to perform aggregation:

- ***the aggregation pipeline***
- the map-reduce function
- single purpose aggregation methods.

Aggregation Pipeline

Documents enter a ***multi-stage*** pipeline that transforms the documents into aggregated results. For example:



First Stage: The $match stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".

Second Stage: The $group stage groups the documents by the cust_id field to calculate the sum of the amount for each unique cust_id.

About the Pipeline

The MongoDB aggregation pipeline consists of stages. ***Each stage*** transforms the documents as they pass through the pipeline.

Pipeline stages do ***not need to produce*** one output document for every input document; e.g., some stages may generate new documents or filter out documents.

Pipeline stages can appear ***multiple times*** in the pipeline with the exception of $out, $merge, and $geoNear stages.

For a list of all available stages, see: https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/#aggregation-pipeline-operator-reference.

Aggregation Pipeline Mapping to SQL functions

SQL Terms	MongoDB Agg	Explanation
WHERE	$match	Filter documents
GROUP BY	$group	Group documents by value, summarize documents. Applies accumulator expression to each group
HAVING	$match	Filters documents with respect to specific criteria that are passed on to next stage of pipeline
SELECT	$project	Reshape documents, include exclude fields, create new fields
ORDER BY	$sort	Reorder document with respect to specific sort key
SUM()	$sum	Returns sum of each group. Ignores non-numeric values
COUNT()	$sum	See above
join	$lookup	Performs left outer join
N/A	$unwind	Deconstructs an array field and returns a document for each array element

Aggregation with the Zip Code Data Set

The examples in this class use the zipcodes collection.

This collection is available at: ***media.mongodb.org/zips.json***.
Use mongoimport to load this data set into your mongod instance.

Each document in the zipcodes collection has the following form:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

Return States with Populations above 10 Million (I)

The aggregation below shows a possible solution:

```
> db.zipcodes.aggregate( [
...     { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
...     { $match: { totalPop: { $gte: 10*1000*1000 } } }
... ] )
{ "_id" : "TX", "totalPop" : 16984601 }
{ "_id" : "FL", "totalPop" : 12686644 }
{ "_id" : "IL", "totalPop" : 11427576 }
{ "_id" : "OH", "totalPop" : 10846517 }
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "PA", "totalPop" : 11881643 }
```


```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```


An original document

Return States with Populations above 10 Million (II)

The ***$group stage*** groups the documents of the zipcode collection by the state field, calculates the totalPop field for each state, ***and outputs a document for each unique state***.

```
> db.zipcodes.aggregate( [    { $group: { _id: "$state", totalPop: { $sum: "$pop" } } }])
{ "_id" : "WI", "totalPop" : 4891769 }
{ "_id" : "WV", "totalPop" : 1793146 }
{ "_id" : "MA", "totalPop" : 6016425 }
{ "_id" : "SC", "totalPop" : 3486703 }
{ "_id" : "ND", "totalPop" : 638272 }
{ "_id" : "LA", "totalPop" : 4217595 }
{ "_id" : "NV", "totalPop" : 1201833 }
{ "_id" : "WA", "totalPop" : 4866692 }
{ "_id" : "NH", "totalPop" : 1109252 }
{ "_id" : "GA", "totalPop" : 6478216 }
{ "_id" : "OK", "totalPop" : 3145585 }
{ "_id" : "OH", "totalPop" : 10846517 }
{ "_id" : "MT", "totalPop" : 798948 }
{ "_id" : "DC", "totalPop" : 606900 }
{ "_id" : "VA", "totalPop" : 6181479 }
{ "_id" : "MI", "totalPop" : 9295297 }
{ "_id" : "MS", "totalPop" : 2573216 }
{ "_id" : "IN", "totalPop" : 5544136 }
{ "_id" : "WY", "totalPop" : 453528 }
{ "_id" : "NM", "totalPop" : 1515069 }
```


```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```


An original document

The new per-state documents have ***two fields***: the *_id* field and the *totalPop* field.

Return States with Populations above 10 Million (III)

The ***$match stage*** filters the grouped documents to output only those documents whose totalPop value is greater than or equal to 10 million.

```
> db.zipcodes.aggregate( [
...     { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
...     { $match: { totalPop: { $gte: 10*1000*1000 } } }
... ] )
{ "_id" : "TX", "totalPop" : 16984601 }
{ "_id" : "FL", "totalPop" : 12686644 }
{ "_id" : "IL", "totalPop" : 11427576 }
{ "_id" : "OH", "totalPop" : 10846517 }
{ "_id" : "CA", "totalPop" : 29754890 }
{ "_id" : "NY", "totalPop" : 17990402 }
{ "_id" : "PA", "totalPop" : 11881643 }
```

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

An original document

The $match stage does not alter the matching documents but outputs the matching documents unmodified.

Return Average City Population by State (I)

In this example, the aggregation pipeline consists of a ***$group stage*** followed by ***another $group stage***:

```
> db.zipcodes.aggregate( [
...     { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
...     { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
... ] )
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NV", "avgCityPop" : 18209.590909090908 }
{ "_id" : "TN", "avgCityPop" : 9656.350495049504 }
{ "_id" : "AR", "avgCityPop" : 4175.355239786856 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
{ "_id" : "NY", "avgCityPop" : 13131.680291970803 }
{ "_id" : "NE", "avgCityPop" : 3034.882692307692 }
{ "_id" : "UT", "avgCityPop" : 9518.508287292818 }
{ "_id" : "CA", "avgCityPop" : 27756.42723880597 }
{ "_id" : "ID", "avgCityPop" : 4320.811158798283 }
{ "_id" : "AZ", "avgCityPop" : 20591.16853932584 }
{ "_id" : "NJ", "avgCityPop" : 15775.89387755102 }
{ "_id" : "TX", "avgCityPop" : 13775.02108678021 }
{ "_id" : "MN", "avgCityPop" : 5372.21375921376 }
{ "_id" : "KY", "avgCityPop" : 4767.164721141375 }
{ "_id" : "IA", "avgCityPop" : 3123.0821147356583 }
{ "_id" : "KS", "avgCityPop" : 3819.884259259259 }
{ "_id" : "CT", "avgCityPop" : 14674.625 }
{ "_id" : "AL", "avgCityPop" : 7907.2152641878665 }
{ "_id" : "PA", "avgCityPop" : 8679.067202337472 }
```

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

An original document

Return Average City Population by State (II)

The first $group stage ***groups the documents*** by the combination of ***city*** and ***state***, uses the ***$sum expression*** to calculate the population for each combination, and ***outputs a document for each city and state combination***.

```
> db.zipcodes.aggregate( [
... ...     { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } }]);
{ "_id" : { "state" : "MA", "city" : "SOUTH WALPOLE" }, "pop" : 752 }
{ "_id" : { "state" : "PA", "city" : "LAKE HARMONY" }, "pop" : 1203 }
{ "_id" : { "state" : "MN", "city" : "QUAMBA" }, "pop" : 1222 }
{ "_id" : { "state" : "AL", "city" : "BEATRICE" }, "pop" : 1620 }
{ "_id" : { "state" : "PA", "city" : "LITTLE MARSH" }, "pop" : 2844 }
{ "_id" : { "state" : "ND", "city" : "CANNON BALL" }, "pop" : 608 }
{ "_id" : { "state" : "MO", "city" : "SOUTH WEST CITY" }, "pop" : 1201 }
{ "_id" : { "state" : "MO", "city" : "MARSHFIELD" }, "pop" : 10026 }
{ "_id" : { "state" : "NE", "city" : "GENEVA" }, "pop" : 3224 }
{ "_id" : { "state" : "AR", "city" : "GOULD" }, "pop" : 3765 }
{ "_id" : { "state" : "MA", "city" : "SPRINGFIELD" }, "pop" : 148062 }
{ "_id" : { "state" : "TX", "city" : "THRALL" }, "pop" : 852 }
{ "_id" : { "state" : "TX", "city" : "EDEN" }, "pop" : 2028 }
{ "_id" : { "state" : "PA", "city" : "IMPERIAL" }, "pop" : 3626 }
{ "_id" : { "state" : "WV", "city" : "APPLE GROVE" }, "pop" : 115 }
{ "_id" : { "state" : "OH", "city" : "SAINT BERNARD" }, "pop" : 8838 }
{ "_id" : { "state" : "OH", "city" : "PETERSBURG" }, "pop" : 650 }
{ "_id" : { "state" : "NC", "city" : "BONNIE DOONE" }, "pop" : 35745 }
{ "_id" : { "state" : "MI", "city" : "UNION CITY" }, "pop" : 3600 }
{ "_id" : { "state" : "MO", "city" : "WALNUT SHADE" }, "pop" : 722 }
```

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

An original document

Return Average City Population by State (III)

The ***second $group stage*** groups the documents in the pipeline by the "_id.state" field (i.e. the state field inside the _id document), uses the ***$avg expression*** to calculate the average city population (***avgCityPop***) for each state, and ***outputs a document for each state***.

```
> db.zipcodes.aggregate( [
...     { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
...     { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
... ] )
{ "_id" : "SD", "avgCityPop" : 1839.6746031746031 }
{ "_id" : "NV", "avgCityPop" : 18209.590909090908 }
{ "_id" : "TN", "avgCityPop" : 9656.350495049504 }
{ "_id" : "AR", "avgCityPop" : 4175.355239786856 }
{ "_id" : "WI", "avgCityPop" : 7323.00748502994 }
{ "_id" : "NY", "avgCityPop" : 13131.680291970803 }
{ "_id" : "NE", "avgCityPop" : 3034.882692307692 }
{ "_id" : "UT", "avgCityPop" : 9518.508287292818 }
{ "_id" : "CA", "avgCityPop" : 27756.42723880597 }
{ "_id" : "ID", "avgCityPop" : 4320.811158798283 }
{ "_id" : "AZ", "avgCityPop" : 20591.16853932584 }
{ "_id" : "NJ", "avgCityPop" : 15775.89387755102 }
{ "_id" : "TX", "avgCityPop" : 13775.02108678021 }
{ "_id" : "MN", "avgCityPop" : 5372.21375921376 }
{ "_id" : "KY", "avgCityPop" : 4767.164721141375 }
{ "_id" : "IA", "avgCityPop" : 3123.0821147356583 }
{ "_id" : "KS", "avgCityPop" : 3819.884259259259 }
{ "_id" : "CT", "avgCityPop" : 14674.625 }
{ "_id" : "AL", "avgCityPop" : 7907.2152641878665 }
{ "_id" : "PA", "avgCityPop" : 8679.067202337472 }
```

Return Largest and Smallest Cities by State

To solve this query we build a pipeline which consists of a ***$group*** stage, a ***$sort*** stage, another ***$group*** stage, and a ***$project*** stage:

Let consider the first two stages:

```
> db.zipcodes.aggregate( [
...     { $group:
...        {
...          _id: { state: "$state", city: "$city" },
...          pop: { $sum: "$pop" }
...        }
...     },
...     { $sort: { pop: 1 } }])
{ "_id" : { "state" : "KS", "city" : "ARNOLD" }, "pop" : 0 }
{ "_id" : { "state" : "TX", "city" : "ECLETO" }, "pop" : 0 }
{ "_id" : { "state" : "NY", "city" : "RAQUETTE LAKE" }, "pop" : 0 }
{ "_id" : { "state" : "HI", "city" : "NINOLE" }, "pop" : 0 }
{ "_id" : { "state" : "OR", "city" : "KENT" }, "pop" : 0 }
{ "_id" : { "state" : "VA", "city" : "WALLOPS ISLAND" }, "pop" : 0 }
{ "_id" : { "state" : "LA", "city" : "FORDOCHE" }, "pop" : 0 }
{ "_id" : { "state" : "AK", "city" : "CHEVAK" }, "pop" : 0 }
{ "_id" : { "state" : "VT", "city" : "UNIV OF VERMONT" }, "pop" : 0 }
{ "_id" : { "state" : "NM", "city" : "KIRTLAND A F B E" }, "pop" : 0 }
{ "_id" : { "state" : "KY", "city" : "TATEVILLE" }, "pop" : 0 }
{ "_id" : { "state" : "NM", "city" : "REGINA" }, "pop" : 0 }
{ "_id" : { "state" : "AK", "city" : "EMMONAK" }, "pop" : 0 }
{ "_id" : { "state" : "WV", "city" : "MOUNT CARBON" }, "pop" : 0 }
{ "_id" : { "state" : "NM", "city" : "ALGODONES" }, "pop" : 0 }
{ "_id" : { "state" : "NC", "city" : "GLOUCESTER" }, "pop" : 0 }
{ "_id" : { "state" : "AK", "city" : "RUSSIAN MISSION" }, "pop" : 0 }
{ "_id" : { "state" : "CA", "city" : "ALLEGHANY" }, "pop" : 0 }
{ "_id" : { "state" : "CA", "city" : "VINTON" }, "pop" : 0 }
{ "_id" : { "state" : "NY", "city" : "CHILDWOLD" }, "pop" : 0 }
Type "it" for more
```

The ***first $group stage*** groups the documents by the combination of the city and state, calculates the sum of the pop values for each combination, and ***outputs a document for each city and state combination.***

The ***$sort stage*** orders the documents in the pipeline by the pop field value, from smallest to largest; i.e. by increasing order. This operation ***does not alter*** the documents.

Return Largest and Smallest Cities by State

Let consider also the ***second $group stage*** in the pipeline:

```
> db.zipcodes.aggregate( [
...     { $group:
...        {
...          _id: { state: "$state", city: "$city" },
...          pop: { $sum: "$pop" }
...        }
...     },
...     { $sort: { pop: 1 } },
...     { $group:
...        {
...          _id : "$_id.state",
...          biggestCity:  { $last: "$_id.city" },
...          biggestPop:   { $last: "$pop" },
...          smallestCity: { $first: "$_id.city" },
...          smallestPop:  { $first: "$pop" }
...        }
...     }])
```

This stage groups the sorted documents by the _id.state field and outputs a document for each state.

Using the ***$last*** ($first) expression, the ***$group*** operator creates the *biggestCity* (smallestCity) and *biggestPop* (smallestPop) fields.

```
{ "_id" : "RI", "biggestCity" : "CRANSTON", "biggestPop" : 176404, "smallestCity" : "CLAYVILLE", "smallestPop" : 45 }
{ "_id" : "CT", "biggestCity" : "BRIDGEPORT", "biggestPop" : 141638, "smallestCity" : "EAST KILLINGLY", "smallestPop" : 25 }
{ "_id" : "KS", "biggestCity" : "WICHITA", "biggestPop" : 295115, "smallestCity" : "ARNOLD", "smallestPop" : 0 }
{ "_id" : "NC", "biggestCity" : "CHARLOTTE", "biggestPop" : 465833, "smallestCity" : "GLOUCESTER", "smallestPop" : 0 }
{ "_id" : "PA", "biggestCity" : "PHILADELPHIA", "biggestPop" : 1610956, "smallestCity" : "HAMILTON", "smallestPop" : 0 }
{ "_id" : "AL", "biggestCity" : "BIRMINGHAM", "biggestPop" : 242606, "smallestCity" : "ALLEN", "smallestPop" : 0 }
{ "_id" : "IA", "biggestCity" : "DES MOINES", "biggestPop" : 148155, "smallestCity" : "DOUDS", "smallestPop" : 15 }
{ "_id" : "MS", "biggestCity" : "JACKSON", "biggestPop" : 204788, "smallestCity" : "CHUNKY", "smallestPop" : 79 }
{ "_id" : "DE", "biggestCity" : "NEWARK", "biggestPop" : 111674, "smallestCity" : "BETHEL", "smallestPop" : 108 }
{ "_id" : "IN", "biggestCity" : "INDIANAPOLIS", "biggestPop" : 348868, "smallestCity" : "WESTPOINT", "smallestPop" : 145 }
{ "_id" : "AZ", "biggestCity" : "PHOENIX", "biggestPop" : 890853, "smallestCity" : "HUALAPAI", "smallestPop" : 2 }
{ "_id" : "ID", "biggestCity" : "BOISE", "biggestPop" : 165522, "smallestCity" : "KEUTERVILLE", "smallestPop" : 0 }
{ "_id" : "KY", "biggestCity" : "LOUISVILLE", "biggestPop" : 288058, "smallestCity" : "TATEVILLE", "smallestPop" : 0 }
{ "_id" : "VT", "biggestCity" : "BURLINGTON", "biggestPop" : 39127, "smallestCity" : "UNIV OF VERMONT", "smallestPop" : 0 }
{ "_id" : "TX", "biggestCity" : "HOUSTON", "biggestPop" : 2095918, "smallestCity" : "ECLETO", "smallestPop" : 0 }
{ "_id" : "CO", "biggestCity" : "DENVER", "biggestPop" : 451182, "smallestCity" : "CHEYENNE MTN AFB", "smallestPop" : 0 }
{ "_id" : "MN", "biggestCity" : "MINNEAPOLIS", "biggestPop" : 344719, "smallestCity" : "JOHNSON", "smallestPop" : 12 }
{ "_id" : "NJ", "biggestCity" : "NEWARK", "biggestPop" : 275572, "smallestCity" : "IMLAYSTOWN", "smallestPop" : 17 }
{ "_id" : "NY", "biggestCity" : "BROOKLYN", "biggestPop" : 2300504, "smallestCity" : "RAQUETTE LAKE", "smallestPop" : 0 }
{ "_id" : "MD", "biggestCity" : "BALTIMORE", "biggestPop" : 733081, "smallestCity" : "ANNAPOLIS JUNCTI", "smallestPop" : 32 }
Type "it" for more
```

Return Largest and Smallest Cities by State

```
> db.zipcodes.aggregate( [
...    { $group:
...       {
...         _id: { state: "$state", city: "$city" },
...         pop: { $sum: "$pop" }
...       }
...    },
...    { $sort: { pop: 1 } },
...    { $group:
...       {
...         _id : "$_id.state",
...         biggestCity:  { $last: "$_id.city" },
...         biggestPop:   { $last: "$pop" },
...         smallestCity: { $first: "$_id.city" },
...         smallestPop:  { $first: "$pop" }
...       }
...    },
...
...   // the following $project is optional, and
...   // modifies the output format.
...
...   { $project:
...     { _id: 0,
...       state: "$_id",
...       biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
...       smallestCity: { name: "$smallestCity", pop: "$smallestPop" }
...     }
...   }
... ] )
{ "biggestCity" : { "name" : "LOUISVILLE", "pop" : 288058 }, "smallestCity" : { "name" : "TATEVILLE", "pop" : 0 }, "state" : "KY" }
{ "biggestCity" : { "name" : "BURLINGTON", "pop" : 39127 }, "smallestCity" : { "name" : "UNIV OF VERMONT", "pop" : 0 }, "state" : "VT" }
{ "biggestCity" : { "name" : "HOUSTON", "pop" : 2095918 }, "smallestCity" : { "name" : "ECLETO", "pop" : 0 }, "state" : "TX" }
{ "biggestCity" : { "name" : "DENVER", "pop" : 451182 }, "smallestCity" : { "name" : "CHEYENNE MTN AFB", "pop" : 0 }, "state" : "CO" }
{ "biggestCity" : { "name" : "MINNEAPOLIS", "pop" : 344719 }, "smallestCity" : { "name" : "JOHNSON", "pop" : 12 }, "state" : "MN" }
{ "biggestCity" : { "name" : "NEWARK", "pop" : 275572 }, "smallestCity" : { "name" : "IMLAYSTOWN", "pop" : 17 }, "state" : "NJ" }
```

The ***final $project stage:*** i) renames the _id field to state and ii) moves the biggestCity, biggestPop, smallestCity, and smallestPop into biggestCity and smallestCity ***embedded documents***.

Exercises (Projections)

Consider the restaurant collections imported as an exercise during the previous class

1. Write a MongoDB query to display the fields restaurant_id, name, borough and cuisine for all the documents in the collection restaurant.

2. Write a MongoDB query to display the fields restaurant_id, name, borough and cuisine, but exclude the field _id for all the documents in the collection restaurant.

3. Write a MongoDB query to display the fields restaurant_id, name, borough and zip code, but exclude the field _id for all the documents in the collection restaurant.

4. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'ces' as last three letters for its name.

Exercises (Projections)

5. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which belong to the borough Staten Island or Queens or Bronxor Brooklyn.

6. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which achieved a score which is not more than 10.

7. Write a MongoDB query to find the restaurant Id, name, and grades for those restaurants which achieved a grade of "A" and scored 11 on an ISODate "2014-08-11T00:00:00Z" among many of survey dates.

8. Write a MongoDB query to know whether all the addresses contains the street or not.

9.Write a MongoDB query which will select all documents in the restaurants collection where the coord field value is double.

Exercises (Aggregations)

1. Propose a couple of Analytics for the Restaurant Dataset that can be implemented using MongoDB aggregations

2. Implement the defined Analytics using MongoDB

Suggested Readings

Students are invited to read the official documentation of MongoDB.

The documentation is available at:
https://docs.mongodb.com/manual/tutorial/project-fields-from-query-results/
https://docs.mongodb.com/manual/tutorial/query-for-null-fields/
https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/
https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Students are also invited to repeat all the examples on their MongoDB shell.