SESSION 4
AGGREGATION PIPELINE

Learning Objectives

In this session, students will learn to:

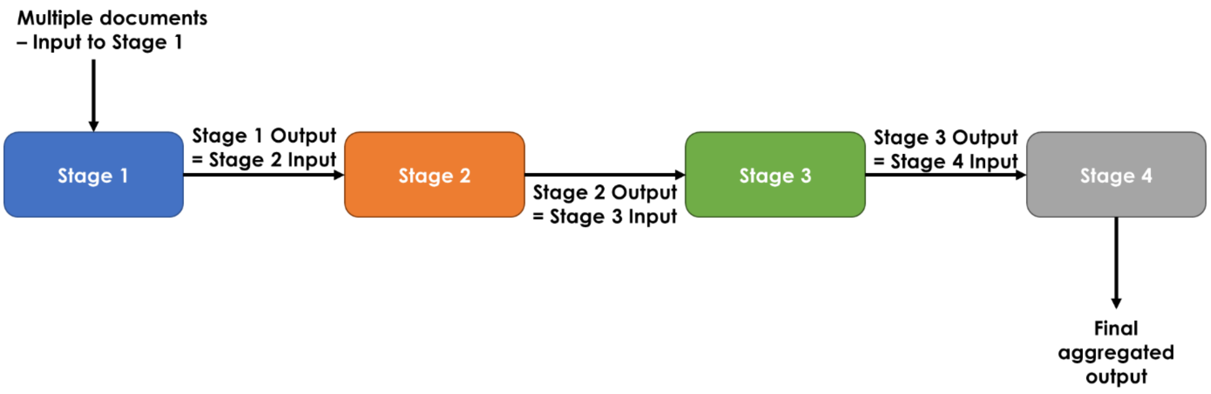
- Explain aggregation pipeline in MongoDB
- Describe the stages in the aggregation pipeline
- Explain the expressions that can be used in the aggregation pipeline

Aggregation in MongoDB helps in grouping multiple documents, applying some processes on them, and returning a single result. This helps in getting useful insights that aid in business decisions. One of the methods of performing data aggregation in MongoDB is the aggregation pipeline.

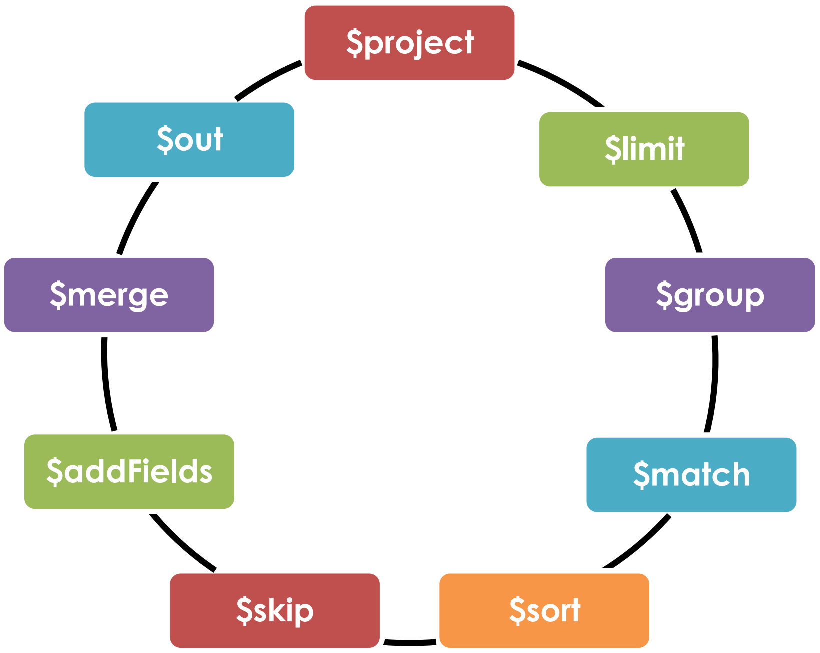
This session will explain what aggregation pipeline in MongoDB is. It will discuss various stages in the aggregation pipeline. It will also explain the various expressions that can be used with the aggregation pipeline.

4.1 Aggregation Pipeline Stages

In the context of MongoDB, a pipeline is nothing, but a series of stages involved in data processing. In the pipeline, the first stage takes documents from the database as input, performs one or more calculations to produce a result. This result is then passed as input to the next stage in the pipeline. This continues until the last stage of pipeline, which finally delivers the desired aggregated output.



The pipeline stages exist in an array in the `db.collection.aggregate` method. The aggregation pipeline can consist of many stages. This session will discuss the most important nine stages of the aggregation pipeline:



4.1.1 `$project` Stage

The `$project` stage collects the required documents, manipulates the documents by adding new fields, deleting existing fields, or adding computed fields. It then passes these modified documents as input to the next stage. The format of `$project` is:

```
{$project: {<specification(s)>}}
```

In this format, <specification (s)> can take one of the values as given in Table 4.1.

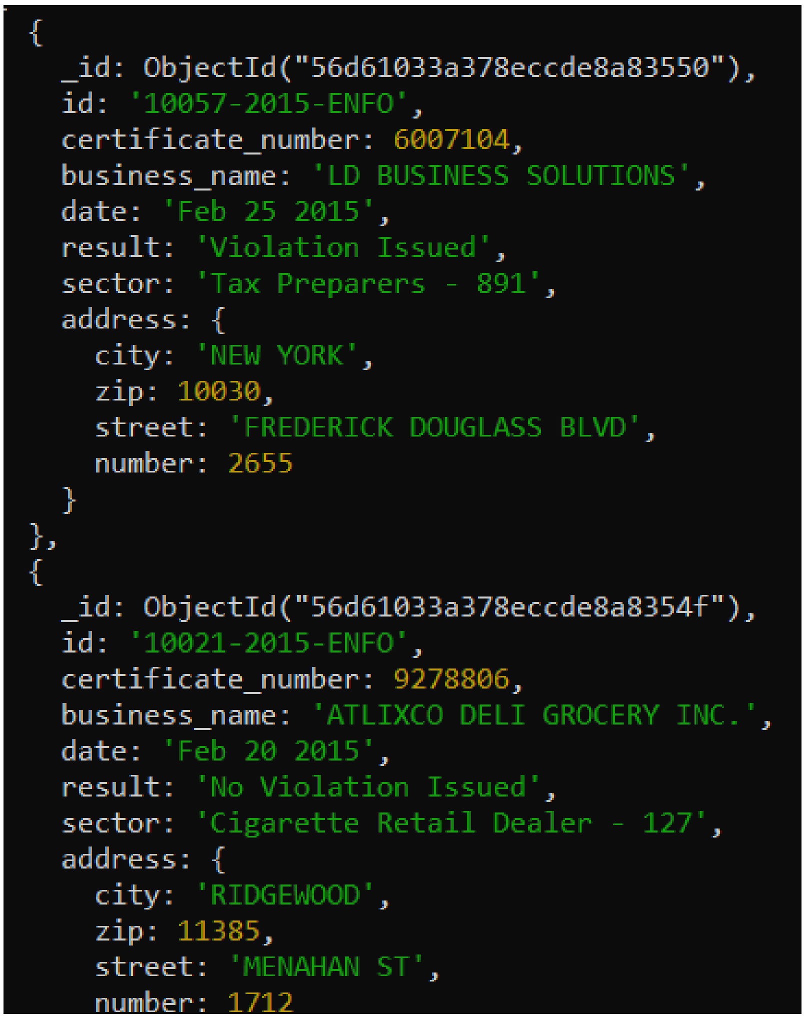
Value	Description	Example
`field: 0 or false`	The field is not included in the result.	`{_id: 0}` The default field, `_id` is not included in the result.
`field: 1 or true`	The field is included in the result.	`{certificate_number: 1}` The `certificate_number` field is included in the result.
`new field: expression`	A new field with the specified expression is included in the result.	`{grade: 'Grade field'}` A new field called `grade` is included in the result with value as `Grade field`.

Table 4.1: Values of `<specification(s)>` in the `$project` Stage

For example, consider that the user wants to aggregate data present in the `inspections` collection from the `sample_training` database. To view the existing data in the `inspections` collection, the user runs the query as:

```
db.inspections.find()
```

Figure 4.1 shows the output of this query.



```
{
  _id: ObjectId("56d61033a378eccde8a83550"),
  id: '10057-2015-ENFO',
  certificate_number: 6007104,
  business_name: 'LD BUSINESS SOLUTIONS',
  date: 'Feb 25 2015',
  result: 'Violation Issued',
  sector: 'Tax Preparers - 891',
  address: {
    city: 'NEW YORK',
    zip: 10030,
    street: 'FREDERICK DOUGLASS BLVD',
    number: 2655
  }
},
{
  _id: ObjectId("56d61033a378eccde8a8354f"),
  id: '10021-2015-ENFO',
  certificate_number: 9278806,
  business_name: 'ATLIXCO DELI GROCERY INC.',
  date: 'Feb 20 2015',
  result: 'No Violation Issued',
  sector: 'Cigarette Retail Dealer - 127',
  address: {
    city: 'RIDGEWOOD',
    zip: 11385,
    street: 'MENAHAN ST',
    number: 1712
```

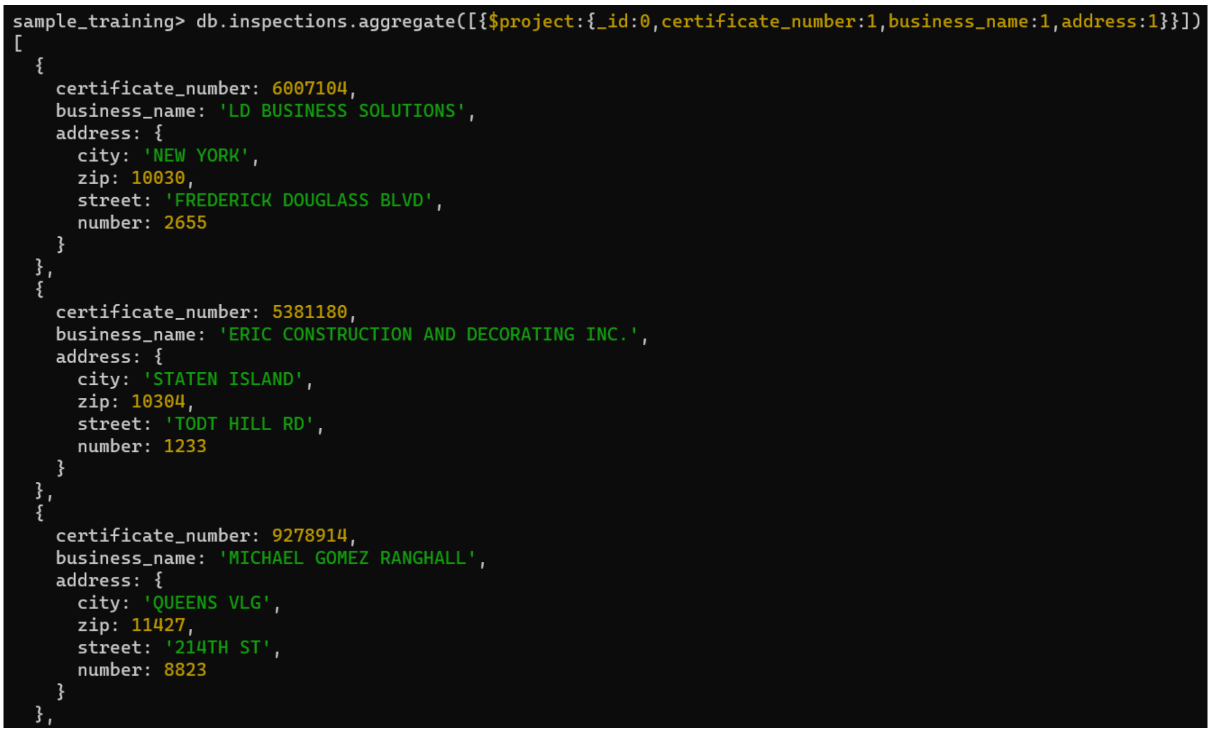
Figure 4.1: Data in the `inspections` Collection

Now, consider that the user wants to view only the `certificate_number`, `business_name`, and `address` fields. In this case, the user can use the `$project` stage of the aggregation pipeline as:

```
db.inspections.aggregate( [ { $project : {
certificate_number : 1 , business_name : 1,
address:1 } } ] )
```

The `_id` filed is displayed by default. So, the user must exclude this field by setting it to `0`.

Figure 4.2 shows the output of the query.



```
sample_training> db.inspections.aggregate([{$project:{_id:0,certificate_number:1,business_name:1,address:1}}])
[
  {
    certificate_number: 6007104,
    business_name: 'LD BUSINESS SOLUTIONS',
    address: {
      city: 'NEW YORK',
      zip: 10030,
      street: 'FREDERICK DOUGLASS BLVD',
      number: 2655
    }
  },
  {
    certificate_number: 5381180,
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: {
      city: 'STATEN ISLAND',
      zip: 10304,
      street: 'TODT HILL RD',
      number: 1233
    }
  },
  {
    certificate_number: 9278914,
    business_name: 'MICHAEL GOMEZ RANGHALL',
    address: {
      city: 'QUEENS VLG',
      zip: 11427,
      street: '214TH ST',
      number: 8823
    }
  },
```

Figure 4.2: Result of the `$project` Stage

The output returns all the documents with the specified fields.

4.1.2 `$limit` Stage

The `$limit` stage helps in limiting the number of documents that are passed to the next stage. The format of the `$limit` stage is:

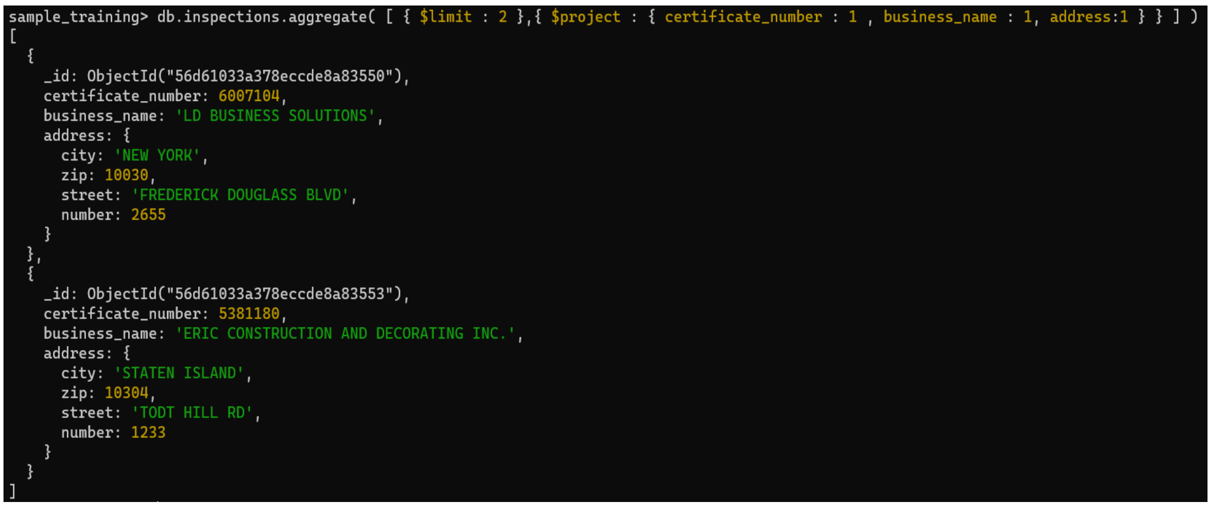
```
{$limit:  {<64-bit positive integer>}}
```

The number of documents that are passed to the next stage is controlled by the number specified in `{<64-bit positive integer>}}`.

Consider the `inspections` collection. To limit the number of documents returned to two and display only specific fields, the `$limit` stage can be used with the `$project` stage as:

```
db.inspections.aggregate( [ { $limit : 2
},{ $project : { certificate_number : 1 ,
business_name : 1, address:1 } } ] )
```

Figure 4.3 shows the output of the query.



```
sample_training> db.inspections.aggregate( [ { $limit : 2 },{ $project : { certificate_number : 1 , business_name : 1, address:1 } } ] )
[
  {
    _id: ObjectId("56d61033a378eccde8a83550"),
    certificate_number: 6007104,
    business_name: 'LD BUSINESS SOLUTIONS',
    address: {
      city: 'NEW YORK',
      zip: 10030,
      street: 'FREDERICK DOUGLASS BLVD',
      number: 2655
    }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83553"),
    certificate_number: 5381180,
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: {
      city: 'STATEN ISLAND',
      zip: 10304,
      street: 'TODT HILL RD',
      number: 1233
    }
  }
]
```

Figure 4.3: Result of the `$limit` Stage

4.1.3 `$group` Stage

The `$group` stage helps in grouping the documents based on a field or key or group of fields. The format of the `$group` stage is:

```
{$group: {_id: <expression>,//Group Key
<field1>: {<accumulator1>: <expression1>}, … }}
```

Parameters in this format are described in Table 4.2.

Specification Form	Description
`_id: <expression>`	Specifies the field based on which the data must be grouped
`<field1>`	Specifies the name of the field whose values must be used for computation
`<accumulator1>`	Specifies the function to be applied on the field specified in `<field1>`

Table 4.2: Parameters of `$group` stage

The `<accumulator1>` parameter can specify one of the functions listed in Table 4.3.

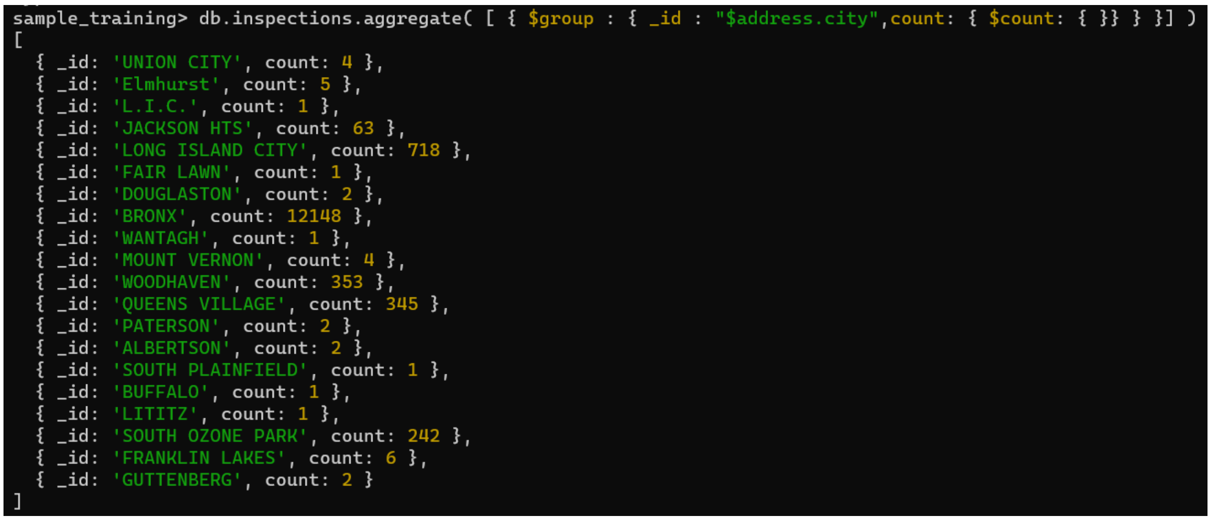
Function	Description
`$avg`	Used to calculate the average value of the group
`$bottom`	Used to retrieve the bottom element in the group based on the order in which the elements are sorted
`$bottomN`	Used to retrieve the bottom N element in the group based on the order in which the elements are sorted
`$top`	Used to retrieve the top element of the group based on the order in which the elements are sorted
`$topN`	Used to retrieve the top N element in the group based on the order in which the elements are sorted
`$min`	Used to retrieve the element with the lowest value in the group
`$minN`	Used to retrieve N elements with the lowest value in the group
`$max`	Used to retrieve the element with the highest value in the group
`$mergeObjects`	Returns a document created by combining the input documents for each group
`$first`	Used to retrieve the value from the first document in each group
`$last`	Used to retrieve the value from the last document in each group
`$count`	Used to know the number of documents in each group
`$sum`	Used to calculate the sum of the numerical values of a field in a group

Table 4.3: Accumulator Functions

For example, consider that in the `inspections` collection, the user wants to group the document based on the `address.city` field and find the number of documents in each group. To do so, the user can run the query as:

```
db.inspections.aggregate( [ { $group : { _id :
"$address.city",count: { $count: { }} } }] )
```

Figure 4.4 shows the output of this query.



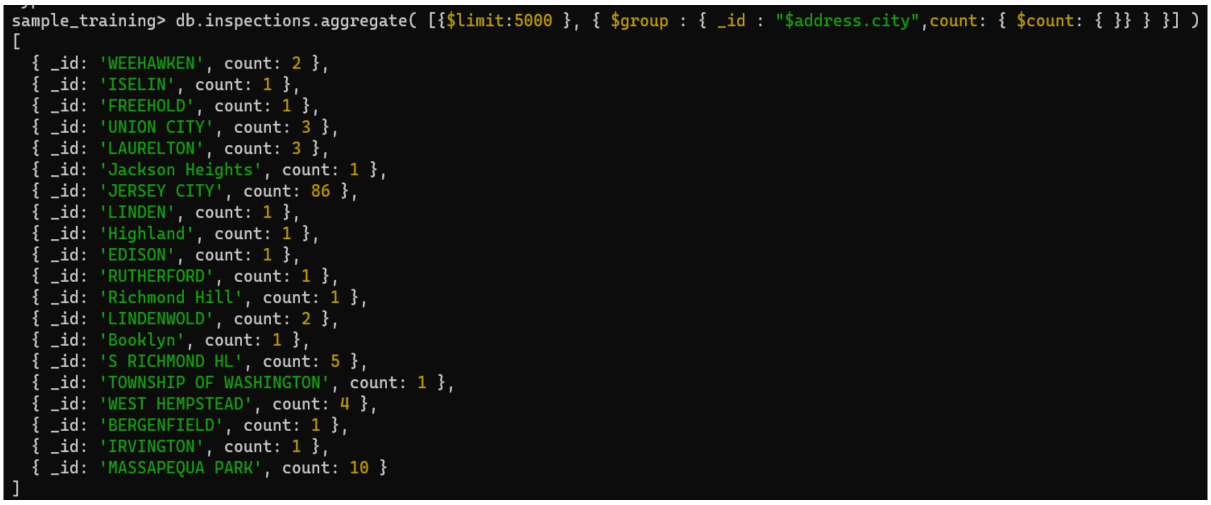
```
sample_training> db.inspections.aggregate( [ { $group : { _id : "$address.city",count: { $count: { }} } }] )
[
  { _id: 'UNION CITY', count: 4 },
  { _id: 'Elmhurst', count: 5 },
  { _id: 'L.I.C.', count: 1 },
  { _id: 'JACKSON HTS', count: 63 },
  { _id: 'LONG ISLAND CITY', count: 718 },
  { _id: 'FAIR LAWN', count: 1 },
  { _id: 'DOUGLASTON', count: 2 },
  { _id: 'BRONX', count: 12148 },
  { _id: 'WANTAGH', count: 1 },
  { _id: 'MOUNT VERNON', count: 4 },
  { _id: 'WOODHAVEN', count: 353 },
  { _id: 'QUEENS VILLAGE', count: 345 },
  { _id: 'PATERSON', count: 2 },
  { _id: 'ALBERTSON', count: 2 },
  { _id: 'SOUTH PLAINFIELD', count: 1 },
  { _id: 'BUFFALO', count: 1 },
  { _id: 'LITITZ', count: 1 },
  { _id: 'SOUTH OZONE PARK', count: 242 },
  { _id: 'FRANKLIN LAKES', count: 6 },
  { _id: 'GUTTENBERG', count: 2 }
]
```

Figure 4.4: Result of the `$count` Accumulator in the `$group` Stage

Consider that the user wants to limit the number of documents passed to the `$group` stage to `5000`. The user wants to group the resulting documents based on the `address.city` field. After doing so, the user wants to find the number of documents in each group. To do so, the user can use the query as:

```
db.inspections.aggregate( [{$limit:5000 }, { $group :
{ _id : "$address.city",count: { $count: { }} } }] )
```

Figure 4.5 shows the output of this query.



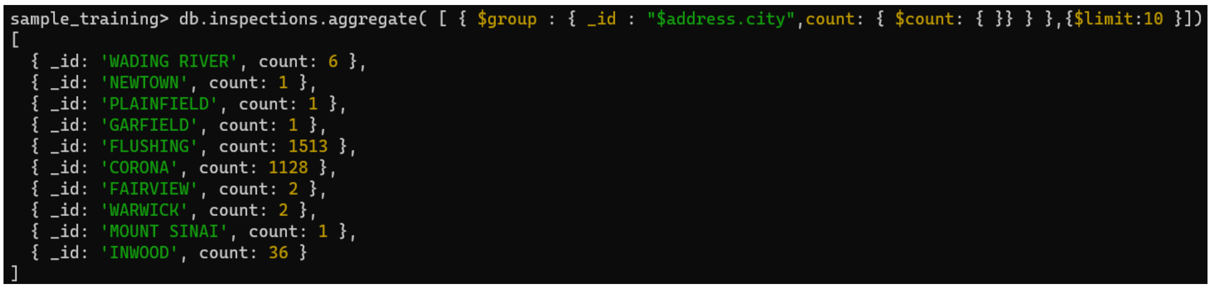
```
sample_training> db.inspections.aggregate( [{$limit:5000 }, { $group : { _id : "$address.city",count: { $count: { }} } }] )
[
  { _id: 'WEEHAWKEN', count: 2 },
  { _id: 'ISELIN', count: 1 },
  { _id: 'FREEHOLD', count: 1 },
  { _id: 'UNION CITY', count: 3 },
  { _id: 'LAURELTON', count: 3 },
  { _id: 'Jackson Heights', count: 1 },
  { _id: 'JERSEY CITY', count: 86 },
  { _id: 'LINDEN', count: 1 },
  { _id: 'Highland', count: 1 },
  { _id: 'EDISON', count: 1 },
  { _id: 'RUTHERFORD', count: 1 },
  { _id: 'Richmond Hill', count: 1 },
  { _id: 'LINDENWOLD', count: 2 },
  { _id: 'Booklyn', count: 1 },
  { _id: 'S RICHMOND HL', count: 5 },
  { _id: 'TOWNSHIP OF WASHINGTON', count: 1 },
  { _id: 'WEST HEMPSTEAD', count: 4 },
  { _id: 'BERGENFIELD', count: 1 },
  { _id: 'IRVINGTON', count: 1 },
  { _id: 'MASSAPEQUA PARK', count: 10 }
]
```

Figure 4.5: Result of `$group` Stage With `$limit` Stage

The user can also use the `$group` and `$limit` stage in the reverse order. That is, if the user wants to group all the documents based on the `address.city` field and count the number of documents in each group. If the user wants to view output only for 10 groups, the user can run the query as:

```
db.inspections.aggregate( [ { $group : { _id :
"$address.city",count: { $count: { }} } },{$limit:10 }])
)db.inspections.aggregate( [ { $group : { _id :
"$address.city",count: { $count: { }} } },{$limit:20 }]
)
```

Figure 4.6 shows the output of this query.



```
sample_training> db.inspections.aggregate( [ { $group : { _id : "$address.city",count: { $count: { }} } },{$limit:10 }])
[
  { _id: 'WADING RIVER', count: 6 },
  { _id: 'NEWTOWN', count: 1 },
  { _id: 'PLAINFIELD', count: 1 },
  { _id: 'GARFIELD', count: 1 },
  { _id: 'FLUSHING', count: 1513 },
  { _id: 'CORONA', count: 1128 },
  { _id: 'FAIRVIEW', count: 2 },
  { _id: 'WARWICK', count: 2 },
  { _id: 'MOUNT SINAI', count: 1 },
  { _id: 'INWOOD', count: 36 }
]
```

Figure 4.6: Result of `$group` Stage With `$limit` Stage

The output will change depending on the order in which the stages are used.

4.1.4 `$match` Stage

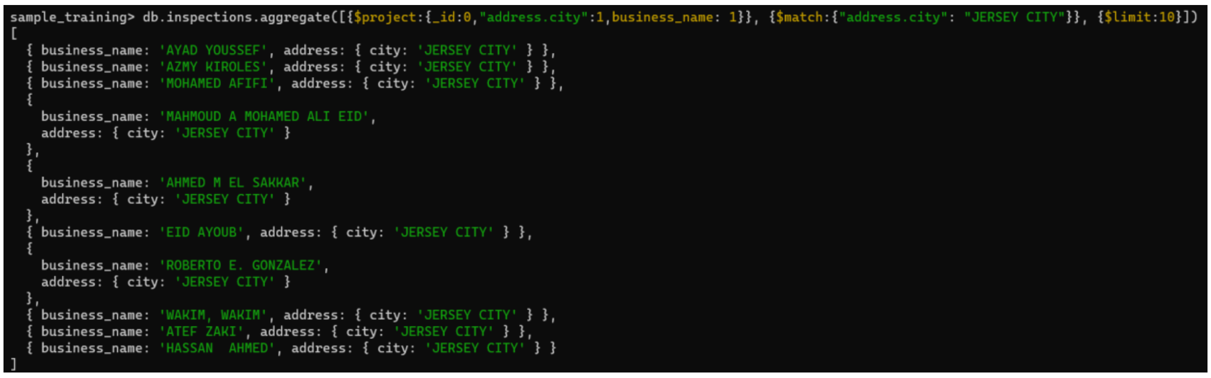
The `$match` stage in the aggregate pipeline helps in filtering documents based on a condition and passing the result to the next stage. The format of `$match` stage is:

```
{$match:{<expression>}}… }}
```

The `$match` takes documents as input and filters the documents on the specified expression. Consider that in the `inspections` collection, the user wants to hide the `_id` field, view the `business_name` and `address.city` fields for documents where the `address.city` is `JERSEY CITY`. Also, the user wants to limit this output to 10 documents. To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.
city":1,business_name: 1}}, {$match:{"address.city":
"JERSEY CITY"}}, {$limit:10}]))
```

Figure 4.7 shows the output of this query.



```
sample_training> db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$match:{"address.city": "JERSEY CITY"}}, {$limit:10}])
[
  { business_name: 'AYAD YOUSSEF', address: { city: 'JERSEY CITY' } },
  { business_name: 'AZMY KIROLES', address: { city: 'JERSEY CITY' } },
  { business_name: 'MOHAMED AFIFI', address: { city: 'JERSEY CITY' } },
  {
    business_name: 'MAHMOUD A MOHAMED ALI EID',
    address: { city: 'JERSEY CITY' }
  },
  {
    business_name: 'AHMED M EL SAKKAR',
    address: { city: 'JERSEY CITY' }
  },
  { business_name: 'EID AYOUB', address: { city: 'JERSEY CITY' } },
  {
    business_name: 'ROBERTO E. GONZALEZ',
    address: { city: 'JERSEY CITY' }
  },
  { business_name: 'WAKIM, WAKIM', address: { city: 'JERSEY CITY' } },
  { business_name: 'ATEF ZAKI', address: { city: 'JERSEY CITY' } },
  { business_name: 'HASSAN  AHMED', address: { city: 'JERSEY CITY' } }
]
```

Figure 4.7: Result of `$match` Stage With `$limit` and `$project` Stages

4.1.5 `$sort` Stage

As the name suggests, the `$sort` stage sorts the documents in the ascending or descending order based on the specified field or fields. The sorted list of documents is then passed to the next stage. If the documents are to be sorted on multiple fields, they are followed in the order of left to right. The format of `$sort` is:

```
{$sort:{field1:<sort order>,field2:<sort order>,…}}
```

Table 4.4 describes the parameters of the `$sort` stage format.

Parameter	Description
`field1`, `field2`	Specifies the names of the fields based on which the documents must be sorted
`sort order`	Specifies the order in which the documents must be sorted. It takes the value 1 for ascending order and -1 for descending order

Table 4.4: Parameters of `$sort` stage

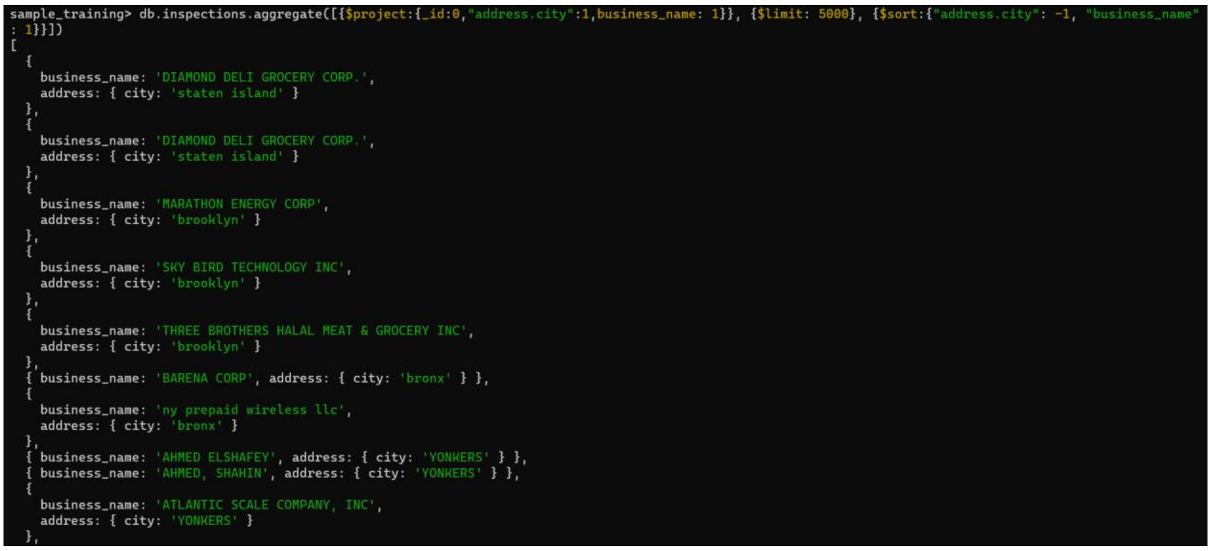
For example, consider that in the `inspections` collection, the user wants to:

- Use 5000 documents
- Hide the `_id` field
- View the `business_name` and `address.city` fields
- Sort the documents based first on the descending order of the `address.city` field and then on the ascending order of the `business_name` field
- Output only 10 documents

To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.city":1,
business_name: 1}}, {$limit: 5000}, {$sort:{"address.city":
-1, "business_name": 1}}, {$limit:10}])
```

Figure 4.8 shows the output of this query.



```
sample_training> db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$limit: 5000}, {$sort:{"address.city": -1, "business_name"
: 1}}])
[
  {
    business_name: 'DIAMOND DELI GROCERY CORP.',
    address: { city: 'staten island' }
  },
  {
    business_name: 'DIAMOND DELI GROCERY CORP.',
    address: { city: 'staten island' }
  },
  {
    business_name: 'MARATHON ENERGY CORP',
    address: { city: 'brooklyn' }
  },
  {
    business_name: 'SKY BIRD TECHNOLOGY INC',
    address: { city: 'brooklyn' }
  },
  {
    business_name: 'THREE BROTHERS HALAL MEAT & GROCERY INC',
    address: { city: 'brooklyn' }
  },
  { business_name: 'BARENA CORP', address: { city: 'bronx' } },
  {
    business_name: 'ny prepaid wireless llc',
    address: { city: 'bronx' }
  },
  { business_name: 'AHMED ELSHAFEY', address: { city: 'YONKERS' } },
  { business_name: 'AHMED, SHAHIN', address: { city: 'YONKERS' } },
  {
    business_name: 'ATLANTIC SCALE COMPANY, INC',
    address: { city: 'YONKERS' }
  },
```

Figure 4.8: Result of `$sort` Stage With `$limit` and `$project` Stages

In the output, the `address.city` field is sorted in descending order. So, all the city names in small letters are displayed first in descending order and then the city names in capital letters are displayed in descending order. Two records having the same city name are sorted in ascending order of the business names as specified.

4.1.6 `$skip` Stage

The `$skip` stage is used to remove some of the documents that are passed to the next stage. The format of the `$skip` stage is:

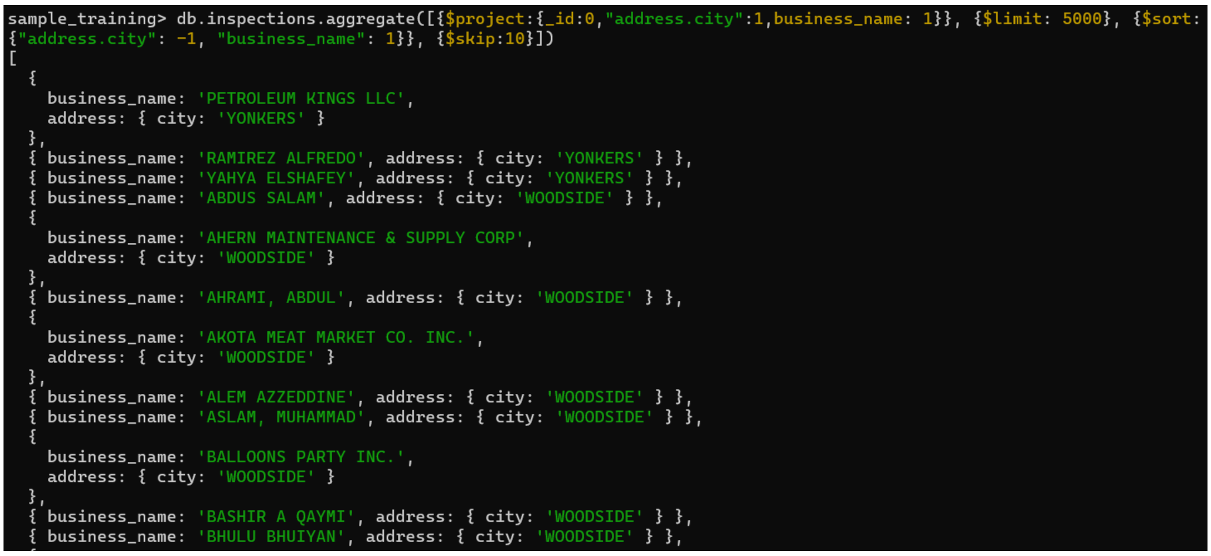
```
{$skip:{<64-bit positive integer}}
```

The number of documents that are passed to the next stage is controlled by the number specified in `{<64-bit positive integer>}}`.

Consider that the user wants to use the same query used for the `$sort` stage. In that query, the user wants to skip the first 10 documents instead of limiting the output to 10 documents. To do so, the user can run the query as:

```
db.inspections.aggregate([{$project:{_id:0,"address.city":1,
business_name: 1}}, {$limit: 5000}, {$sort:{"address.city":
-1, "business_name": 1}}, {$skip:10}])
```

Figure 4.9 shows the output of this query.



```
sample_training> db.inspections.aggregate([{$project:{_id:0,"address.city":1,business_name: 1}}, {$limit: 5000}, {$sort:
{"address.city": -1, "business_name": 1}}, {$skip:10}])
[
  {
    business_name: 'PETROLEUM KINGS LLC',
    address: { city: 'YONKERS' }
  },
  { business_name: 'RAMIREZ ALFREDO', address: { city: 'YONKERS' } },
  { business_name: 'YAHYA ELSHAFEY', address: { city: 'YONKERS' } },
  { business_name: 'ABDUS SALAM', address: { city: 'WOODSIDE' } },
  {
    business_name: 'AHERN MAINTENANCE & SUPPLY CORP',
    address: { city: 'WOODSIDE' }
  },
  { business_name: 'AHRAMI, ABDUL', address: { city: 'WOODSIDE' } },
  {
    business_name: 'AKOTA MEAT MARKET CO. INC.',
    address: { city: 'WOODSIDE' }
  },
  { business_name: 'ALEM AZZEDDINE', address: { city: 'WOODSIDE' } },
  { business_name: 'ASLAM, MUHAMMAD', address: { city: 'WOODSIDE' } },
  {
    business_name: 'BALLOONS PARTY INC.',
    address: { city: 'WOODSIDE' }
  },
  { business_name: 'BASHIR A QAYMI', address: { city: 'WOODSIDE' } },
  { business_name: 'BHULU BHUIYAN', address: { city: 'WOODSIDE' } },
```

Figure 4.9: Result of `$skip` Stage With `$limit`, `$sort`, and `$project` Stages

In the output, the first documents where the values in the `address.city` field were in lower letters have been skipped.

4.1.7 `$addFields` Stage

As the name suggests, the `$addFields` stage is used to add a new field to the documents. It takes the documents passed from the previous stage, adds the new specified field to those documents, and passes the updated documents to the next stage. The format of `$addFields` stage is as follows:

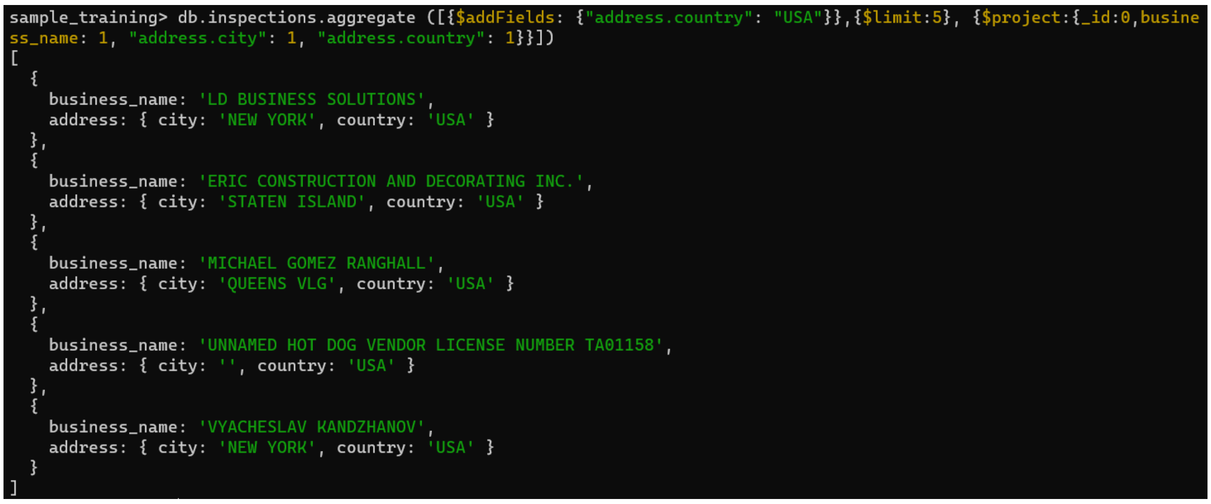
```
{$addFields:{<new field1>:<expression1>….}}
```

In this syntax, `<new field1>` specifies the name for the new field and `<expression1>` specifies the value for the new field. The user can add as many fields as required using a single query.

Consider the `inspections` collection. The user wants to add a new field named `address.country` and assign the value `USA` for this new field. Then, the user wants to display the `business_name`, `address.city`, and `address.country` fields for 5 documents. To do so, the user can run the query as:

```
db.inspections.aggregate ([{$addFields: {"address.country":
"USA"}},{$limit:5}, {$project:{_id:0,business_name: 1,
"address.city": 1, "address.country": 1}}])
```

Figure 4.10 shows the output of this query.



```
sample_training> db.inspections.aggregate ([{$addFields: {"address.country": "USA"}},{$limit:5}, {$project:{_id:0,busine
ss_name: 1, "address.city": 1, "address.country": 1}}])
[
  {
    business_name: 'LD BUSINESS SOLUTIONS',
    address: { city: 'NEW YORK', country: 'USA' }
  },
  {
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: { city: 'STATEN ISLAND', country: 'USA' }
  },
  {
    business_name: 'MICHAEL GOMEZ RANGHALL',
    address: { city: 'QUEENS VLG', country: 'USA' }
  },
  {
    business_name: 'UNNAMED HOT DOG VENDOR LICENSE NUMBER TA01158',
    address: { city: '', country: 'USA' }
  },
  {
    business_name: 'VYACHESLAV KANDZHANOV',
    address: { city: 'NEW YORK', country: 'USA' }
  }
]
```

Figure 4.10: Result of `$addFields` Stage With `$limit`, and `$project` Stages

4.1.8 `$out` Stage

The `$out` stage is used to copy the query results to a collection. If the query results are written to an existing document, the `$out` stage completely overwrites the query results to the collection without retaining any existing documents. The format for `$out` stage is:

```
{ $out: { db: "<output-db>",
coll: "<output-collection>" } }
```

In this format, `db: "<output-db>"` specifies the name of the database where the output collection exists and `coll: "<output-collection>"` specifies the name of the output collection.

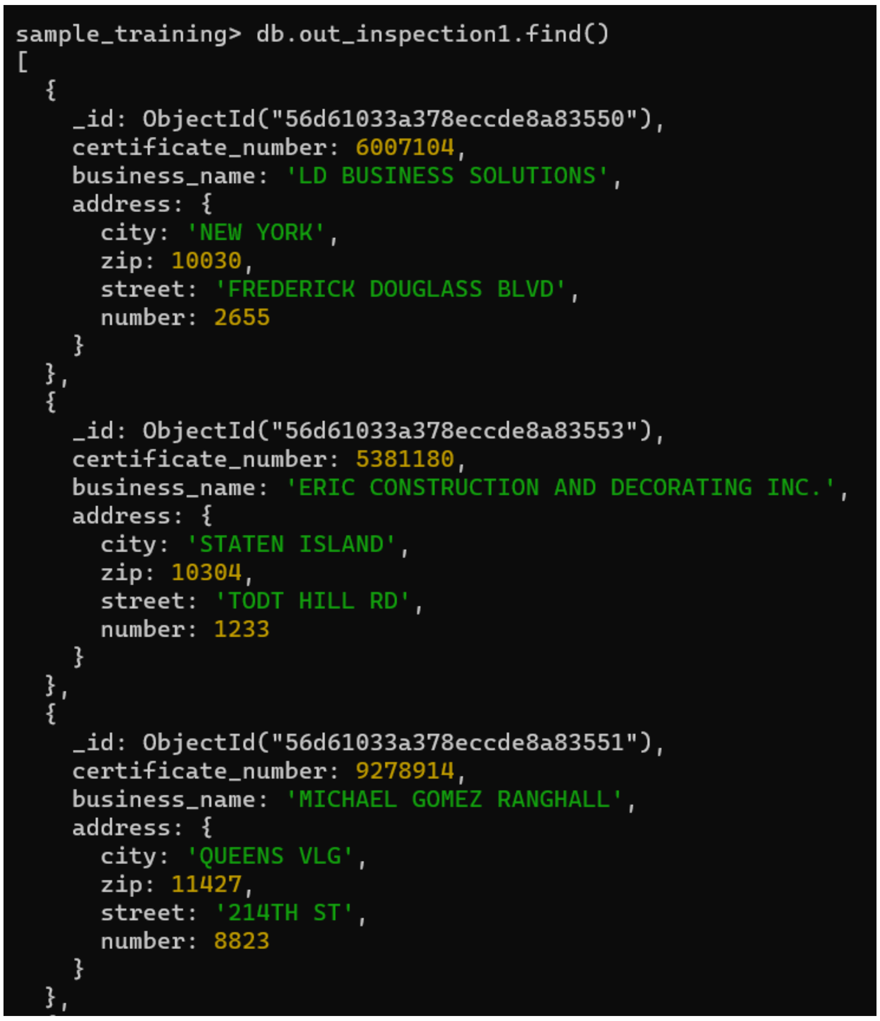
For example, if the user wants to push eight documents from the `inspections` collection with the `certificate_number`, `business_name`, and `address` fields to a new collection named `out_inpsection1`. To do so, the user can run the query as:

```
db.inspections.aggregate( [{$limit:8},{ $project : {
certificate_number : 1 , business_name : 1,
addresaddress:1 } }, { $out : "out_inspection1" } ] )
```

To view the documents in the `out_inpsection1` collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.11 shows the output of this query.



```
sample_training> db.out_inspection1.find()
[
  {
    _id: ObjectId("56d61033a378eccde8a83550"),
    certificate_number: 6007104,
    business_name: 'LD BUSINESS SOLUTIONS',
    address: {
      city: 'NEW YORK',
      zip: 10030,
      street: 'FREDERICK DOUGLASS BLVD',
      number: 2655
    }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83553"),
    certificate_number: 5381180,
    business_name: 'ERIC CONSTRUCTION AND DECORATING INC.',
    address: {
      city: 'STATEN ISLAND',
      zip: 10304,
      street: 'TODT HILL RD',
      number: 1233
    }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83551"),
    certificate_number: 9278914,
    business_name: 'MICHAEL GOMEZ RANGHALL',
    address: {
      city: 'QUEENS VLG',
      zip: 11427,
      street: '214TH ST',
      number: 8823
    }
  },
```

Figure 4.11: Documents in the `out_inspection1` Collection

The `out_inspection1` collection consists of eight documents and the last two documents have the `address.city` as `NEW YORK`.

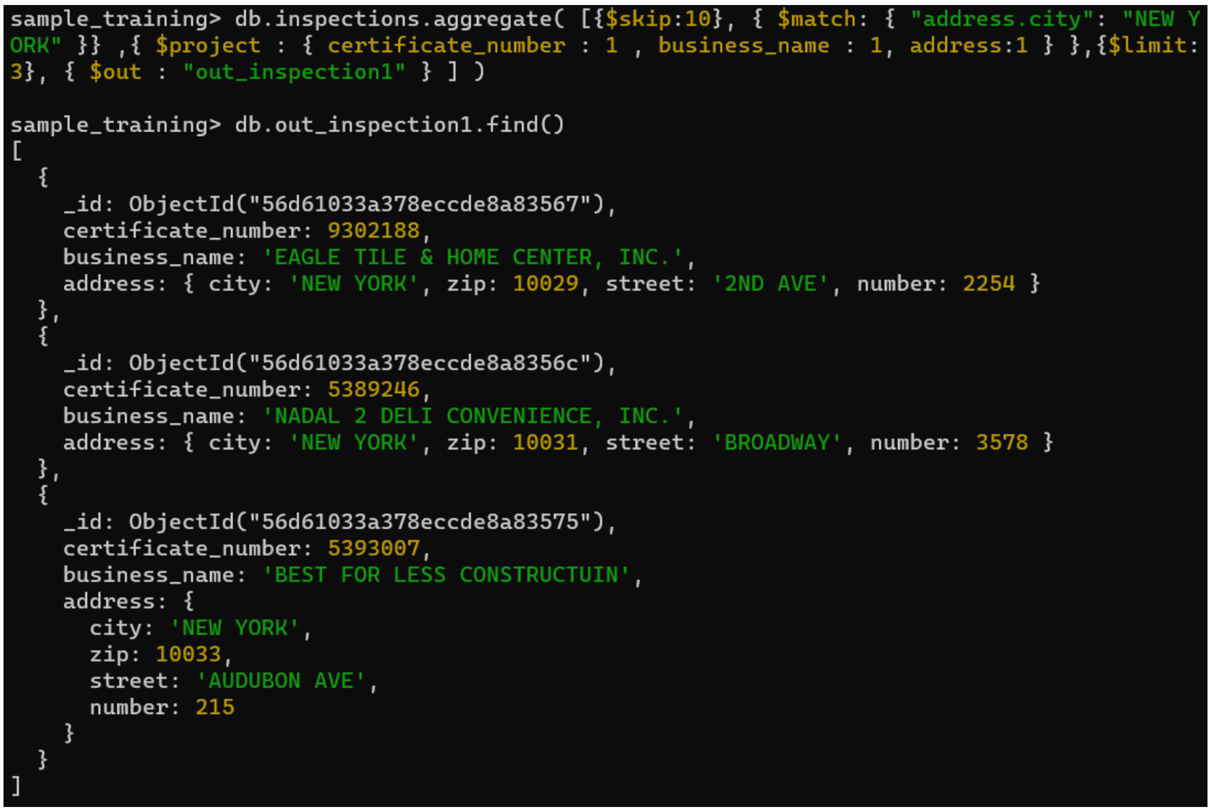
Consider that the user wants to skip the first 10 documents in the `inspections` collection. The user then wants to push three documents with the `certificate_number`, `business_name`, and `address` fields where `address.city` is `NEW YORK` to the `out_inspection1` collection. To do so, the user can run the query as:

```
db.inspections.aggregate( [{$skip:10}, { $match: {
"address.city": "NEW YORK" }} ,{ $project : {
certificate_number : 1 , business_name : 1, address:1 }
},{$limit:3} { $out : "out_inspection" } ] )
```

To view the documents in the `out_inpsection1` collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.12 shows the output of this query.



```
sample_training> db.inspections.aggregate( [{$skip:10}, { $match: { "address.city": "NEW Y
ORK" }} ,{ $project : { certificate_number : 1 , business_name : 1, address:1 } },{$limit:
3}, { $out : "out_inspection1" } ] )

sample_training> db.out_inspection1.find()
[
  {
    _id: ObjectId("56d61033a378eccde8a83567"),
    certificate_number: 9302188,
    business_name: 'EAGLE TILE & HOME CENTER, INC.',
    address: { city: 'NEW YORK', zip: 10029, street: '2ND AVE', number: 2254 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a8356c"),
    certificate_number: 5389246,
    business_name: 'NADAL 2 DELI CONVENIENCE, INC.',
    address: { city: 'NEW YORK', zip: 10031, street: 'BROADWAY', number: 3578 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83575"),
    certificate_number: 5393007,
    business_name: 'BEST FOR LESS CONSTRUCTUIN',
    address: {
      city: 'NEW YORK',
      zip: 10033,
      street: 'AUDUBON AVE',
      number: 215
    }
  }
]
```

Figure 4.12: Documents in the `out_inspection1` Collection

Figure 4.12 shows three documents in `out_inspection1` collection instead of the original eight documents. This is because the `$out` stage has removed the earlier documents and replaced them with the documents from the last query.

4.1.9 `$merge` Stage

The `$merge` stage is similar to the `$out` stage and is used to store the output of the pipeline to a new collection in the same or different database. The difference is that `$merge` does not remove the earlier documents. It uses an identifier that uniquely identifies the document and when a match is found, it performs actions such as replacing existing document, keeping existing document, or merging the documents. If a match is not found, it performs actions such as inserting the document or discarding the document. The format of `$merge` is:

```
{ $merge: {
into: <collection> -or- { db: <db>, coll: <collection> },
on: <identifier field> -or- [ <identifier field1>, ...],
let: <variables>,
whenMatched: <replace|keepExisting|merge|fail|pipeline>,
whenNotMatched: <insert|discard|fail>
} }
```

Table 4.5 describes the parameters used in the `$merge` stage.

Parameter	Description
`into:`	Specifies the collection if the documents are to be pushed into a collection in the same database where the query is being run Otherwise, it specifies the database and collection to which the documents are to be pushed. If the specified collection does not exist, MongoDB creates a new collection.
`on:`	Specifies an identifier that uniquely identifies the documents It is used to identify if a document in the result matches a document in the output collection.
`whenMatched:`	Specifies the action to be taken if the identifier of a document in the output collection matches the identifier of a document in the query result It allows replacing the existing document, retaining the existing document, or merging the documents. When merging the documents, the fields that are missing in the output collection will be added. If the fields exist in both the documents, the values from the query results will replace the values in the output collection. If the identifier in the query results and output collection match, the merge operation can also be failed.

Parameter	Description
`whenNotMatched:`	Specifies the action that must be taken if the identifier of a document in the output collection does not match the identifier of a document in the query result In this case, • the document from the query result can be inserted into the output collection • the documents from the query result can be discarded, or • the merge operation can be failed.

Table 4.5: Parameters of `$merge` stage

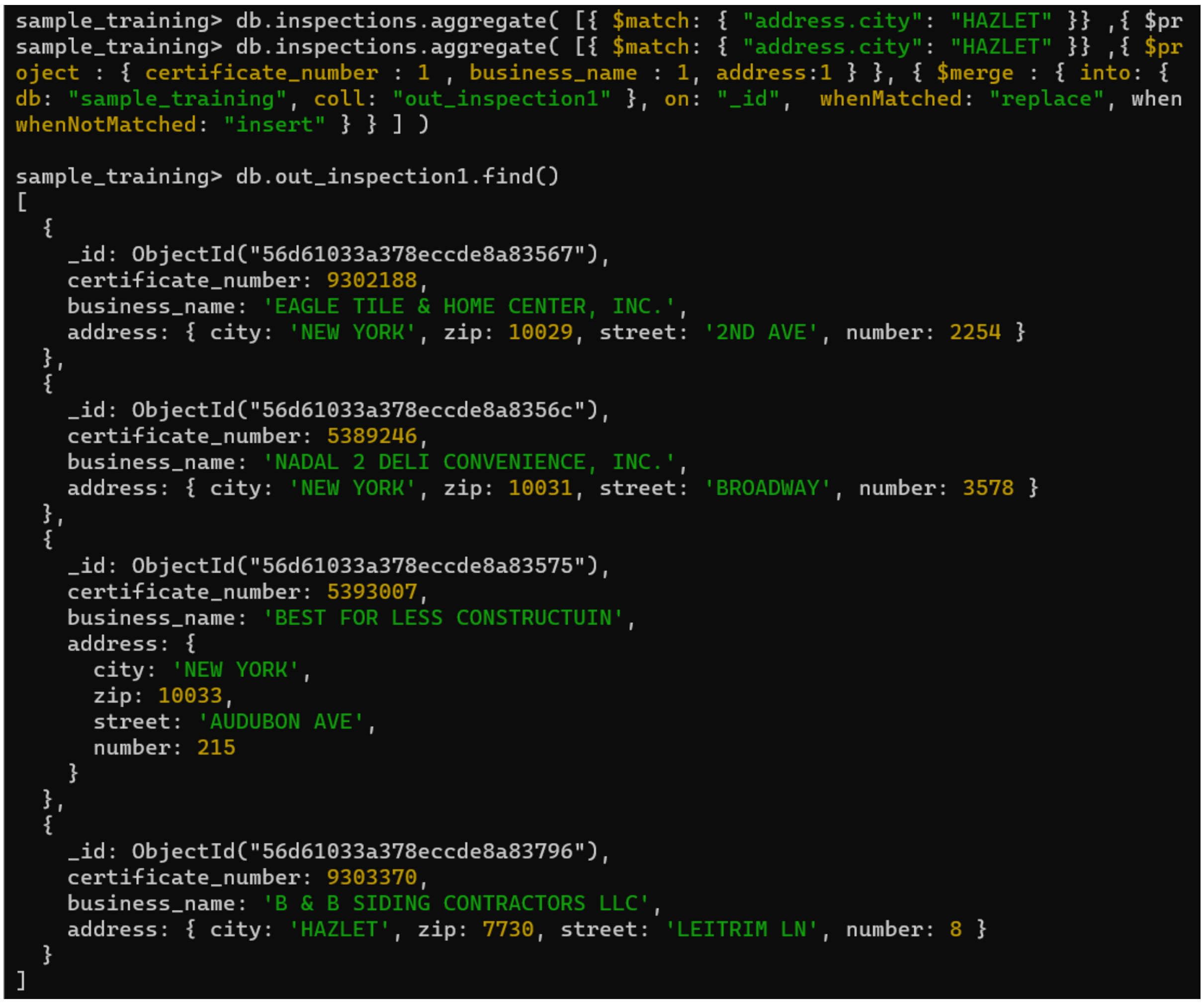
For example, consider that the user wants to push documents from the `inspections` collection into the `out_inspection1` collection where `address.city` is `HAZLET`. If a match is found the documents must be replaced in the `out_inspection1` collection. If a match is not found, the document must be inserted into the `out_inspection1` collection. To do so, the user can run a query as:

```
db.inspections.aggregate( [{$skip:10}, { $match: {
"address.city": "NEW YORK" }} ,{ $project : {
certificate_number : 1 , business_name : 1, address:1 }
},{$limit:3} { $out : "out_inspection" } ] )
```

To view the documents in the `out_inpsection1` collection, the user can run the query as:

```
db.out_inspection1.find()
```

Figure 4.13 shows the output of this query.



```
sample_training> db.inspections.aggregate( [{ $match: { "address.city": "HAZLET" }} ,{ $pr
sample_training> db.inspections.aggregate( [{ $match: { "address.city": "HAZLET" }} ,{ $pr
oject : { certificate_number : 1 , business_name : 1, address:1 } }, { $merge : { into: {
db: "sample_training", coll: "out_inspection1" }, on: "_id",  whenMatched: "replace", when
whenNotMatched: "insert" } } ] )

sample_training> db.out_inspection1.find()
[
  {
    _id: ObjectId("56d61033a378eccde8a83567"),
    certificate_number: 9302188,
    business_name: 'EAGLE TILE & HOME CENTER, INC.',
    address: { city: 'NEW YORK', zip: 10029, street: '2ND AVE', number: 2254 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a8356c"),
    certificate_number: 5389246,
    business_name: 'NADAL 2 DELI CONVENIENCE, INC.',
    address: { city: 'NEW YORK', zip: 10031, street: 'BROADWAY', number: 3578 }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83575"),
    certificate_number: 5393007,
    business_name: 'BEST FOR LESS CONSTRUCTUIN',
    address: {
      city: 'NEW YORK',
      zip: 10033,
      street: 'AUDUBON AVE',
      number: 215
    }
  },
  {
    _id: ObjectId("56d61033a378eccde8a83796"),
    certificate_number: 9303370,
    business_name: 'B & B SIDING CONTRACTORS LLC',
    address: { city: 'HAZLET', zip: 7730, street: 'LEITRIM LN', number: 8 }
  }
]
```

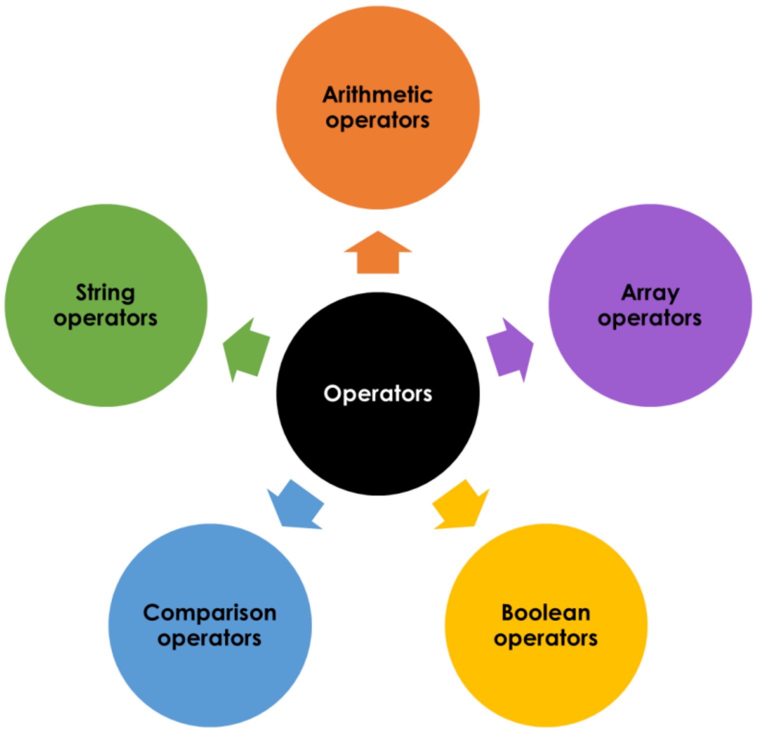
Figure 4.13: Documents in the `out_inspection1` Collection

Figure 4.13 shows that the `out_inspection1` collection now has four documents instead of the original three. This is because the document with `address.city` as `HAZLET` did not find a match and so that document was inserted into the collection.

4.2 Aggregation Pipeline Operators

Aggregation pipeline operators are used in various stages of the aggregation pipeline to perform arithmetic or logical calculations.

Different types of aggregation pipeline operators are:



4.2.1 Arithmetic Operators

Arithmetic operators perform arithmetic operations on the given operands and return a single value. Table 4.6 describes some of the arithmetic operators offered by MongoDB.

Name	Description
`$add`	Used to add two or more numbers or numbers and one date and return the sum. It can only take one input as date. If date is one of the inputs, then the other inputs are treated as milliseconds for addition purposes and the result returned will be a date.
`$subtract`	Used to subtract two specified numbers or dates and return the difference. It accepts two expressions and subtracts the second expression from the first one. If the two values are numbers, it returns the difference in value. If the two values are dates, it returns the difference in milliseconds. If one value is a date and the other is a number, it considers the number as milliseconds and returns the resulting date. In this case, the date should be specified first because it is not meaningful to subtract a date from a number.
`$multiply`	Used to multiply two or more numbers and return the product. It accepts any number of argument expressions.

Name	Description
`$divide`	Used to divide two numbers. It accepts two expressions as input and divides the value of the first expression by the value of the second expression.
`$ceil`	Used to get the smallest integer greater than or equal to the given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is Not a Number, it returns NaN.
`$floor`	Used to get the largest integer less than or equal to the given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is not a number, it returns NaN.
`$abs`	Used to evaluate the absolute value of a given expression, if the expression resolves to a number. If the expression resolves to a null, this operator returns a null. If the expression evaluates to a value that is not a number, it returns NaN.
`$pow`	Used to find the value of the specified number when raised to the power of the specified exponent.
`$exp`	Used to evaluate the value of Euler's number (e) to the specified exponent, if the exponent resolves to a number. If the exponent resolves to a null or a value that is not a number, this operator returns null.
`$ln`	Used to evaluate the natural log of a number.
`$log`	Used to evaluate the log of a number in the specified base.
`$log10`	Used to evaluate the log of a number to the base of 10.
`$mod`	Used to retrieve the remainder of division of two numbers. It accepts two expressions and divides the first number by the second to return the result.
`$round`	Used to round off a number to a whole integer *or* to a specified number of decimal places.
`$sqrt`	Used to calculates the square root of the specified number.
`$trunc`	Used to truncate a number to a whole integer or to a specified number of decimal places.

Table 4.6: Arithmetic Operators

Let us look at an example of one of the arithmetic operators, `$add`. The syntax of the `$add` operator is:

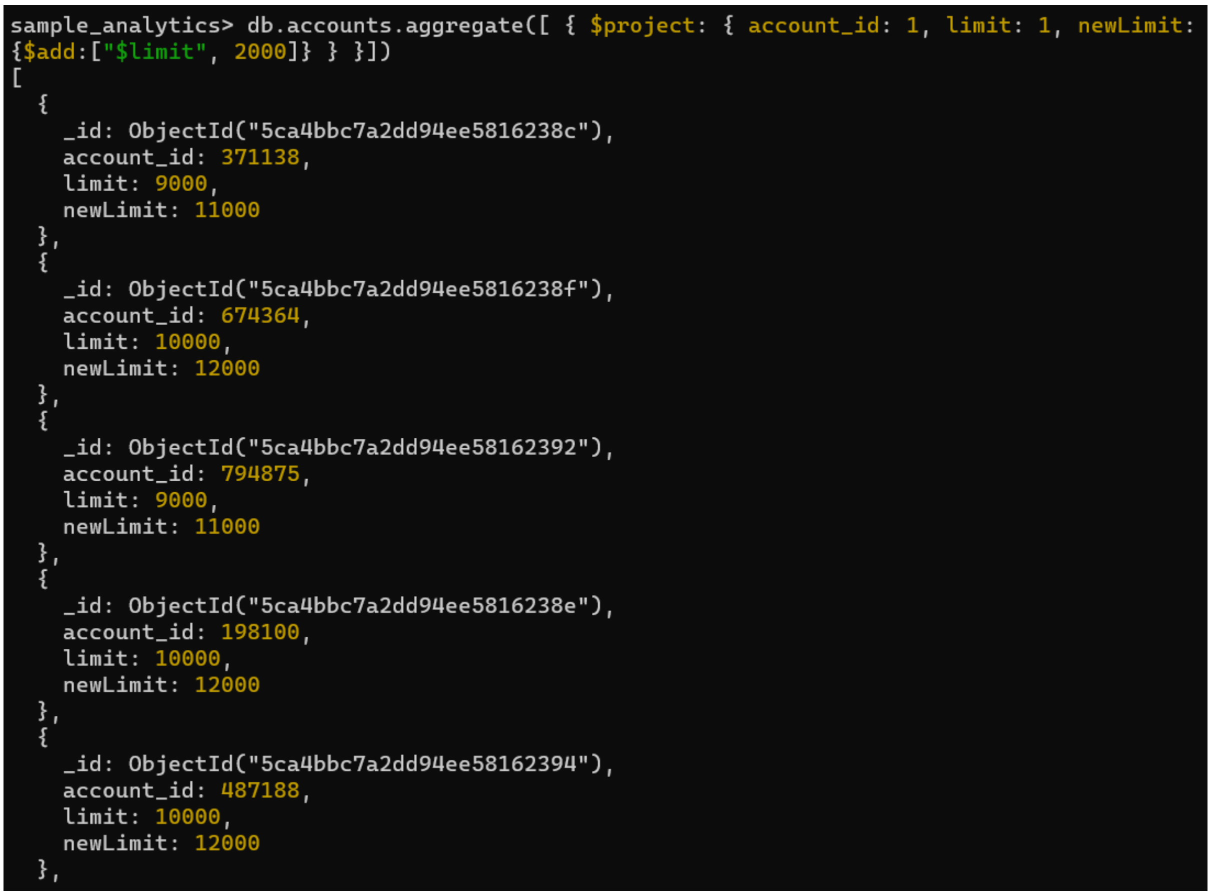
```
$add:{<expression1>,<expression2>,…}
```

In this syntax, `<expression>` can be all valid numbers or a combination of numbers and one date.

Consider that in the `sample_analytics` database, the user wants to increase the sales limit in the `accounts` collection for all entries by 2000. The user wants to store this new limit in a field named `newLimit`. To do so, the user can run a query that uses the `$add` operator as:

```
db.accounts.aggregate([ { $project: { account_id: 1,
limit: 1, newLimit: {$add:["$limit", 2000]} } }])
```

Figure 4.14 shows the output of this query.



```
sample_analytics> db.accounts.aggregate([ { $project: { account_id: 1, limit: 1, newLimit:
{$add:["$limit", 2000]} } }])
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),
    account_id: 371138,
    limit: 9000,
    newLimit: 11000
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
    account_id: 674364,
    limit: 10000,
    newLimit: 12000
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162392"),
    account_id: 794875,
    limit: 9000,
    newLimit: 11000
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
    account_id: 198100,
    limit: 10000,
    newLimit: 12000
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162394"),
    account_id: 487188,
    limit: 10000,
    newLimit: 12000
  },
```

Figure 4.14: Output of Query With `$add` Operator

4.2.2 Array Operators

An array is a collection of values that has a single variable name. Each element in the array is identified by an index, which starts from 0. That is, the first element in the array will have the index as 0. Each value can be referred to by using the array name and its index. MongoDB provides various operators that work on arrays. Table 4.7 describes some of these operators.

Name	Description
`$first`	Used to retrieve the first element in the specified array.
`$firstN`	Used to retrieve the specified number of elements from an array. The elements are retrieved from the start of the array.
`$last`	Used to retrieve the last element in the specified array.
`$lastN`	Used to retrieve the specified number of elements from an array. The elements are retrieved from the end of the array.
`$isArray`	Used to check if the specified expression is an array. If the expression is an array, it returns true; else, it returns false.
`$concatArray`	Used to concatenate multiple arrays. This operator returns a single array.
`$filter`	Used to select specific elements from an array. This operator returns an array with the elements that match the specified condition.
`$sortArray`	Used to sort elements in an array in the ascending or descending order. For ascending order, it takes the sort direction as 1 and for descending order it takes the sort direction as -1.
`$maxN`	Used to retrieve N number of elements with largest values from the array.
`$minN`	Used to retrieve N number of elements with smallest values from the array.
`$arrayElemAt`	Used to retrieve the element located at the specified index in the array.
`$indexofArray`	Used to check if the specified value is present in the specified array. It returns the index of the first occurrence of the value in the array. If the value is not present in the array, it returns the value -1.
`$size`	Used to retrieve the count of the elements in an array. It accepts a single expression as argument.

Name	Description
`$slice`	Used to retrieve a subset of elements in an array from the start or end of the array or from a specified position in the array.
`$arrayToObject`	Used to convert key-value pairs to documents.
`$objecttoArray`	Used to convert a document to an array key-value pairs that represent the documents.
`$in`	Used to check if the specified value is present in the specified array. It returns true if the value is present in the array and false if it is not present in the array.
`$map`	Used to apply an expression to the elements in an array. It returns the array with the new values that are arrived at by applying the expression.
`$range`	Used to create an array with a sequence of numbers that are generated using the specified start number, end number, and step-size.
`$reduce`	Used to apply an expression to each element in an array to arrive at new values. It then combines the values into a single value.
`$reverseArray`	Used to reverse the order of the elements in an array.
`$zip`	Used to get the transpose of two arrays. The resultant arrays will be formed by first elements of the first and second arrays, second elements of the first and second arrays and so on.

Table 4.7 Array Operators

Let us look at an example of the array operators, `$first` and `$last`. The syntax for these operators is:

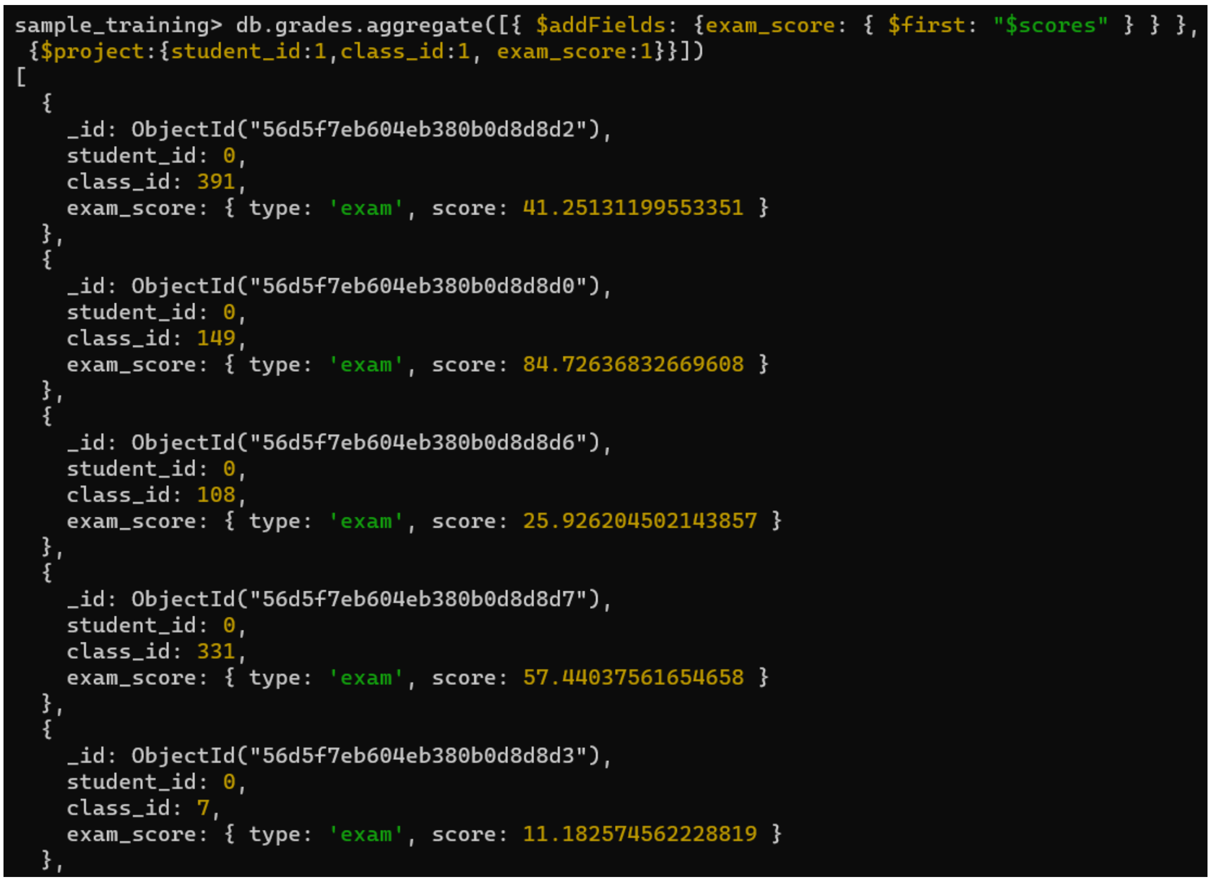
```
$first/$last:<expression>
```

In this syntax, `<expression>` can be any valid expression that resolves to an array.

For example, the `grades` collection in the `sample_training` database has an array named `scores` that provides scores for `exam`, `quiz`, `homework`, and `homework` in that order. Consider that the user wants to retrieve the exam score, which is the first element in the `scores` array. To do so, the user can run a query with the `$first` operator as:

```
db.grades.aggregate([{ $addFields: {exam_score: {
$first: "$scores" } } },
{$project:{student_id:1,class_id:1, exam_score:1}}])
```

Figure 4.15 shows the output of this query.



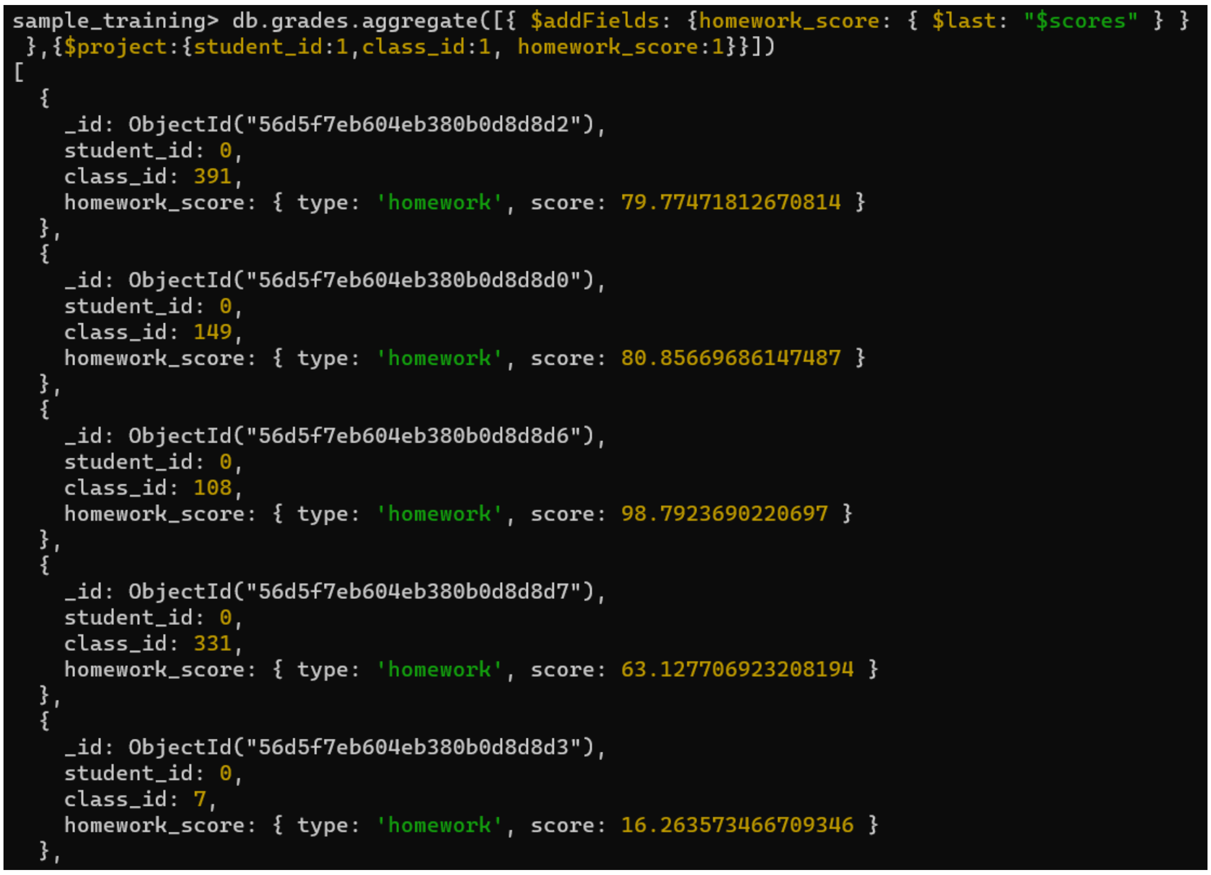
```
sample_training> db.grades.aggregate([{ $addFields: {exam_score: { $first: "$scores" } } },
 {$project:{student_id:1,class_id:1, exam_score:1}}])
[
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d2"),
    student_id: 0,
    class_id: 391,
    exam_score: { type: 'exam', score: 41.25131199553351 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d0"),
    student_id: 0,
    class_id: 149,
    exam_score: { type: 'exam', score: 84.72636832669608 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d6"),
    student_id: 0,
    class_id: 108,
    exam_score: { type: 'exam', score: 25.926204502143857 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d7"),
    student_id: 0,
    class_id: 331,
    exam_score: { type: 'exam', score: 57.44037561654658 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d3"),
    student_id: 0,
    class_id: 7,
    exam_score: { type: 'exam', score: 11.182574562228819 }
  },
```

Figure 4.15: Output of Query with `$first` Operator

Now, consider that the user wants to retrieve the homework score, which is the last element in the `scores` array. To do so, the user can run a query with the `$last` operator as:

```
db.grades.aggregate([{ $addFields:
{homework_score: { $last: "$scores" } }
},{$project:{student_id:1,class_id:1,
homework_score:1}}])
```

Figure 4.16 shows the output of this query.



```
sample_training> db.grades.aggregate([{ $addFields: {homework_score: { $last: "$scores" } }
 },{$project:{student_id:1,class_id:1, homework_score:1}}])
[
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d2"),
    student_id: 0,
    class_id: 391,
    homework_score: { type: 'homework', score: 79.77471812670814 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d0"),
    student_id: 0,
    class_id: 149,
    homework_score: { type: 'homework', score: 80.85669686147487 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d6"),
    student_id: 0,
    class_id: 108,
    homework_score: { type: 'homework', score: 98.7923690220697 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d7"),
    student_id: 0,
    class_id: 331,
    homework_score: { type: 'homework', score: 63.127706923208194 }
  },
  {
    _id: ObjectId("56d5f7eb604eb380b0d8d8d3"),
    student_id: 0,
    class_id: 7,
    homework_score: { type: 'homework', score: 16.263573466709346 }
  },
```

Figure 4.16: Output of Query with `$last` Operator

4.2.3 Boolean Operators

Boolean operator takes the parameters in the form of expression and resolves them to Boolean values. It then returns a Boolean value as described in Table 4.8.

Name	Description
`$and`	Returns true if all the specified expressions resolve to true; else, it returns false. It accepts multiple argument expressions.
`$not`	Returns true if the specified expression resolves to false and it returns false if the specified expression resolves to true. It accepts a single argument expression.
`$or`	Returns true if any one of the specified expressions evaluates to true. It accepts multiple argument expressions.

Table 4.8: Boolean Operators

Let us look at an example of a Boolean operator, `$and`. The syntax for the `$and` operators is:

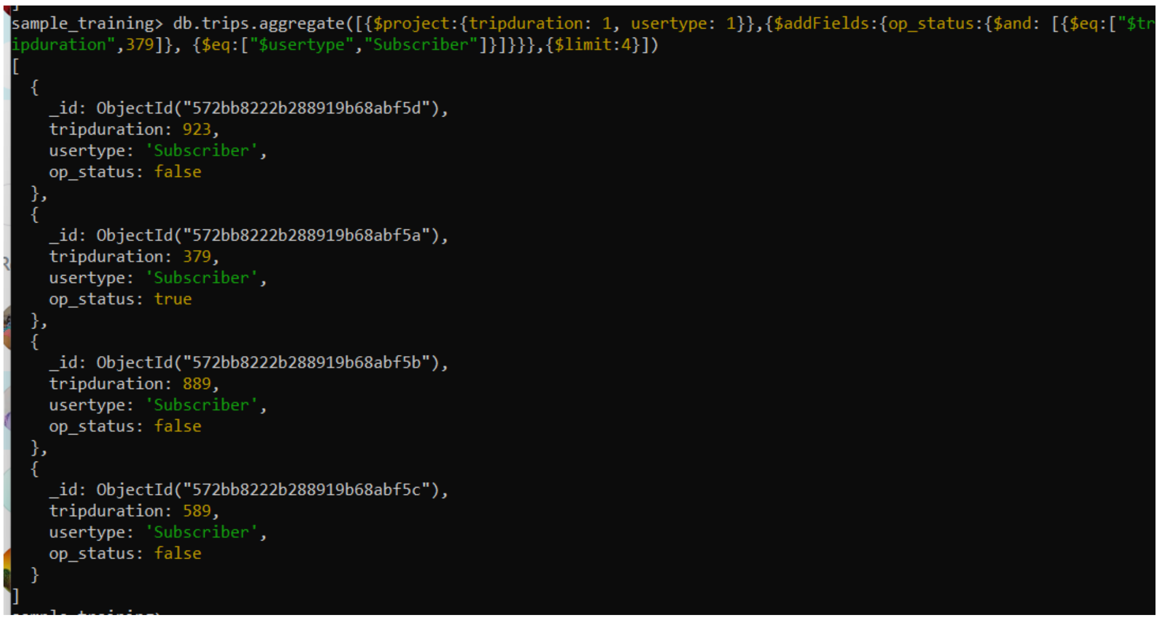
```
{ $and: [ <expression1>, <expression2>, ... ] }
```

In this syntax, `<expression>` can be any valid expression that resolves to a Boolean value.

For example, the `sample_training` database has a `trips` collection, which specifies the `tripduration` and `usertype` for each document. Consider that the user wants to set the value of a new field named `op_status` to `true` for all the documents where `tripduration` is 379 and `usertype` is `Subscriber`. For all the other documents the value of `op_status` will be set to `false`. To do so, the user can run a query with the `$and` operator as:

```
db.trips.aggregate([{$project:{tripduration: 1,
usertype: 1}},{$addFields:{op_status:{$and:
[{$eq:["$tripduration",379]},
{$eq:["$usertype","Subscriber"]}]}}},{$limit:4}])
```

Figure 4.17 shows the output of this query.



```
sample_training> db.trips.aggregate([{$project:{tripduration: 1, usertype: 1}},{$addFields:{op_status:{$and: [{$eq:["$tr
ipduration",379]}, {$eq:["$usertype","Subscriber"]}]}}},{$limit:4}])
[
  {
    _id: ObjectId("572bb8222b288919b68abf5d"),
    tripduration: 923,
    usertype: 'Subscriber',
    op_status: false
  },
  {
    _id: ObjectId("572bb8222b288919b68abf5a"),
    tripduration: 379,
    usertype: 'Subscriber',
    op_status: true
  },
  {
    _id: ObjectId("572bb8222b288919b68abf5b"),
    tripduration: 889,
    usertype: 'Subscriber',
    op_status: false
  },
  {
    _id: ObjectId("572bb8222b288919b68abf5c"),
    tripduration: 589,
    usertype: 'Subscriber',
    op_status: false
  }
]
```

Figure 4.17: Output of Query with `$and` Operator

4.2.4 Comparison Operators

As the name suggests, comparison operators compare two expressions and return a Boolean value. Table 4.9 lists the comparison operators.

Name	Description
`$cmp`	Compares two expressions and returns: • 0 if two expressions are equal • 1 if the first expression is greater than the second one • -1 if the first expression is less than the second one
`$eq`	Compares two expressions and returns true if both the expressions are equal, else, it returns false
`$gt`	Compares two expressions and returns true if the first expression is greater than the second one, else, it returns false
`$gte`	Compares two expressions and returns true if the first expression is greater than or equal to the second one, else, it returns false
`$lt`	Compares two expressions and returns true if the first expression is less than the second one, else, it returns false
`$lte`	Compares two expressions and returns true if the first expression is less than or equal to the second one, else, it returns false
`$ne`	Compares two expressions and returns true if both the expressions are not equal, else, it returns false

Table 4.9: Comparison Operators

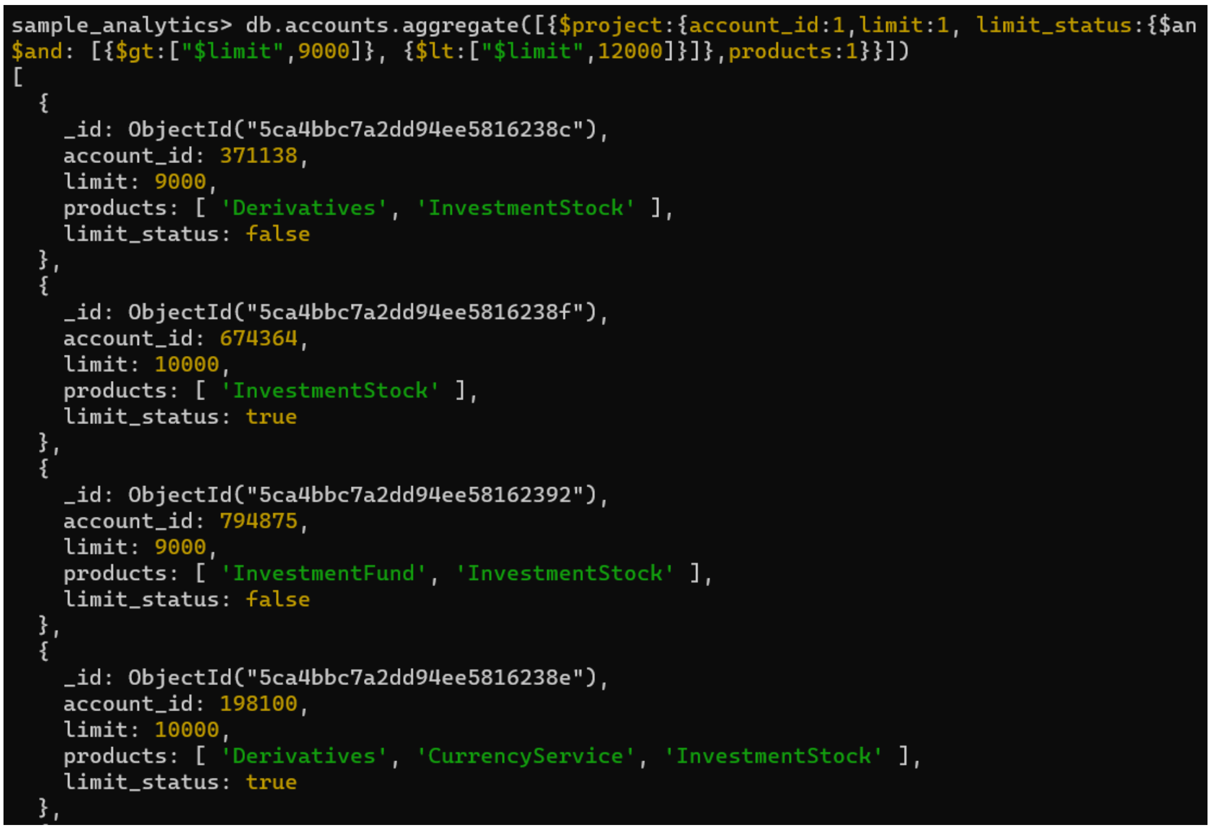
Let us take a look at how to use the comparison operators, `$gt` and `$lt`. The syntax for these operators is:

```
{ $gt/$lt: [ <expression1>, <expression2> ] }
```

For example, the `sample_analytics` database has an `accounts` collection, which specifies the `limit` for each account. Consider that the user wants to retrieve the documents where the `limit` is greater than `9000` and less than `12000`. To do so, the user can run a query with the `$and` operator as:

```
db.accounts.aggregate([{$project:{account_id:1,limit:1,
limit_status:{$and: [{$gt:["$limit",9000]},
{$lt:["$limit",12000]}]},products:1}}])
```

Figure 4.18 shows the output of this query.



```
sample_analytics> db.accounts.aggregate([{$project:{account_id:1,limit:1, limit_status:{$an
$and: [{$gt:["$limit",9000]}, {$lt:["$limit",12000]}]},products:1}}])
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),
    account_id: 371138,
    limit: 9000,
    products: [ 'Derivatives', 'InvestmentStock' ],
    limit_status: false
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
    account_id: 674364,
    limit: 10000,
    products: [ 'InvestmentStock' ],
    limit_status: true
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162392"),
    account_id: 794875,
    limit: 9000,
    products: [ 'InvestmentFund', 'InvestmentStock' ],
    limit_status: false
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
    account_id: 198100,
    limit: 10000,
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ],
    limit_status: true
  },
```

Figure 4.18: Output of Query With `$gt` and `$lt` Operators

4.2.5 String Operator

String operators work on strings. Some of the string operators are listed in Table 4.10.

Name	Description
`$concat`	Used to combine multiple strings into a single string
`$dateFromString`	Used to convert a string into a date object The string that is specified should be a date/time string.
`$datetoString`	Used to convert a date object to a date/time string in the specified format
`$indexofBytes`	Used to search for an occurrence of a substring in a string When it finds the first matching occurrence, it returns the UTF-8 byte index of that occurrence. If it does not find the substring, it returns `-1`.
`$indexofCP`	Used to search for an occurrence of a substring in a string

Name	Description
	When it finds the first matching occurrence, it returns the UTF-8 code point index of that occurrence. If it does not find the substring, it returns -1.
`$ltrim`	Used to remove whitespace or the specified characters from the beginning of a string
`$regexFind`	Used to apply a regular expression (regex) to a string It returns information about the *first* matched substring.
`$regexFindAll`	Used to apply a regular expression (regex) to a string It returns information about all the matched substrings.
`$regexMatch`	Used to apply a regular expression (regex) to a string It returns `true` if a matching substring is found; else, it returns `false`.
`$replaceOne`	Used to replace the first occurrence of a matched string with the given input
`$replaceAll`	Used to replace all the occurrences of a matched string with the given input
`$rtrim`	Used to remove whitespace or the specified characters from the end of a string
`$split`	Used to split a string into substrings based on a delimiter. It returns an array of substrings
`$strLenBytes`	Used to find the length of a string. It returns the number of UTF-8 encoded bytes in a string
`$strLenCP`	Used to find the length of a string. It returns the number of UTF-8 code points in a string
`$strcasecmp`	Used to compare strings based on the casing of the letters in the string It returns: • `0` if the two strings are equivalent • `1` if the first string is greater than the second one • `-1` if the first string is less than the second one
`$toLower`	Used to convert a string to lower case
`$toString`	Used to convert a value to a string
`$trim`	Used to remove whitespace or the specified characters from the beginning and end of a string
`$toUpper`	Used to convert a string to upper case

Table 4.10: String Operators

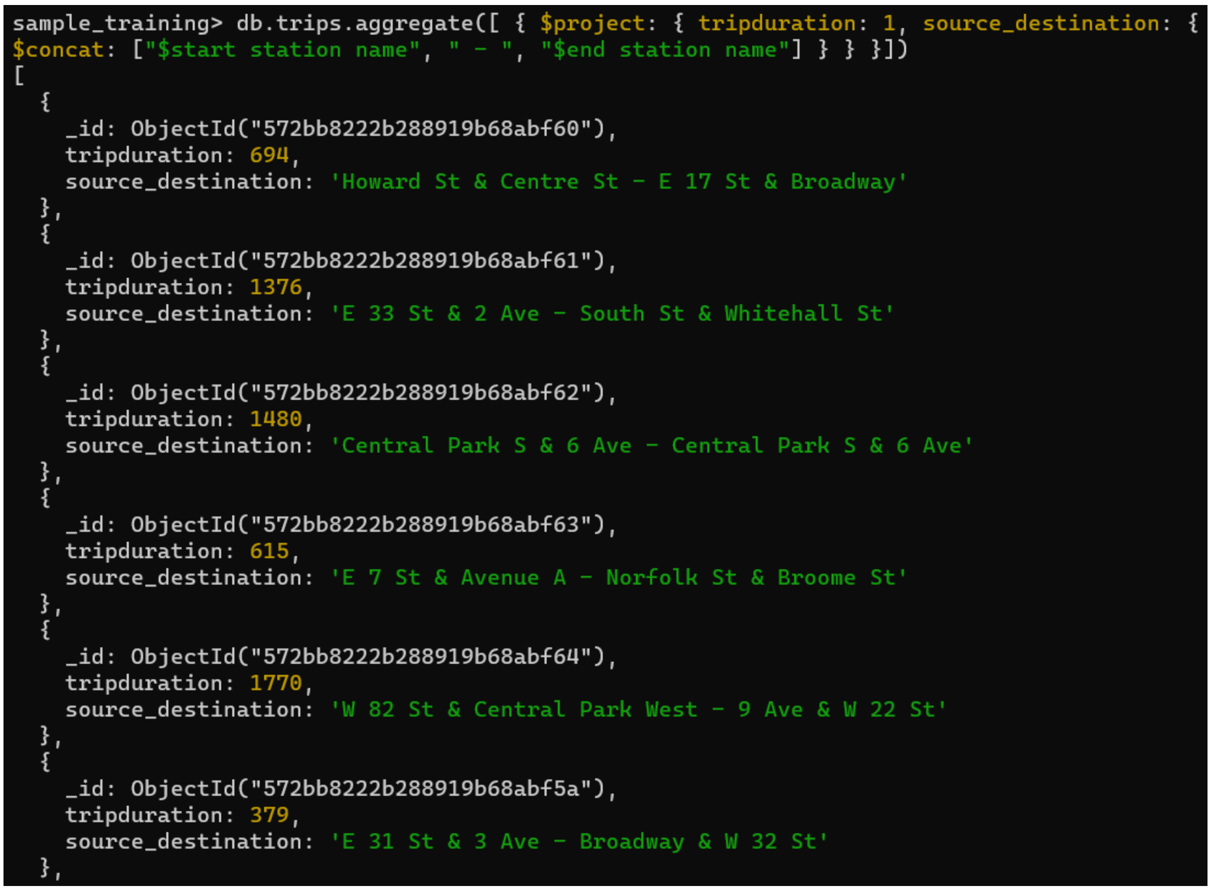
Let us take a look at how to use the string operator, `$concat`. The syntax for this operator is:

```
{ $concat: [ <expression1>, <expression2>, ... ] }
```

For example, the `sample_training` database has a `trips` collection, which specifies the `start station name` and `end station name` for each trip. Consider that the user wants to combine these two names with the delimiter as – and store this value in the `source-destination` field. To do so, the user can run a query with the `$concat` operator as:

```
db.trips.aggregate([ { $project: { tripduration: 1,
source_destination: { $concat: ["$start station
name", " - ", "$end station name"] } } }])
```

Figure 4.19 shows the output of this query.



```
sample_training> db.trips.aggregate([ { $project: { tripduration: 1, source_destination: {
$concat: ["$start station name", " - ", "$end station name"] } } }])
[
  {
    _id: ObjectId("572bb8222b288919b68abf60"),
    tripduration: 694,
    source_destination: 'Howard St & Centre St - E 17 St & Broadway'
  },
  {
    _id: ObjectId("572bb8222b288919b68abf61"),
    tripduration: 1376,
    source_destination: 'E 33 St & 2 Ave - South St & Whitehall St'
  },
  {
    _id: ObjectId("572bb8222b288919b68abf62"),
    tripduration: 1480,
    source_destination: 'Central Park S & 6 Ave - Central Park S & 6 Ave'
  },
  {
    _id: ObjectId("572bb8222b288919b68abf63"),
    tripduration: 615,
    source_destination: 'E 7 St & Avenue A - Norfolk St & Broome St'
  },
  {
    _id: ObjectId("572bb8222b288919b68abf64"),
    tripduration: 1770,
    source_destination: 'W 82 St & Central Park West - 9 Ave & W 22 St'
  },
  {
    _id: ObjectId("572bb8222b288919b68abf5a"),
    tripduration: 379,
    source_destination: 'E 31 St & 3 Ave - Broadway & W 32 St'
  },
```

Figure 4.19: Output of Query with `$concat` Operator

4.3 Summary

- Aggregation is processing of many documents in a collection and giving out a result.
- Aggregation pipeline is a series of steps, where some processing is done at each step and the result is passed to the next stage.
- Some of the stages of the aggregation pipeline include `$project`, `$limit`, `$match`, `$skip`, `$group`, `$sort`, `$addFields`, `$out`, and `$merge`.
- Arithmetic operators, Boolean operators, string operators, and comparison operators are some of the operators in MongoDB that are used to manipulate data in the aggregation pipeline.

Test Your Knowledge

1. Which of the following statement is true about the aggregation pipeline in MongoDB?

   a. It refers to a specific flow of operations that processes, transforms, and returns results.
   b. It consists of one or more stages that process documents.
   c. Each stage takes input from the previous stage.
   d. Documents pass through the stages in random order.

2. Which of the following stage is usually the last stage of an aggregate pipeline?

   a. `$group`
   b. `$sort`
   c. `$merge`
   d. `$skip`

3. Consider that you have a collection `emp_detail` which consists of 20 documents. You must display only eight documents leaving the first two documents. Which of the following options will perform the given task using the `$limit` and `$skip` stages in any order?

   a. `db.emp_detail.aggregate([{$limit: 10},{$skip: 2}])`
   b. `db.emp_detail.aggregate([{$skip: 2},{$limit: 8}])`
   c. `db.emp_detail.aggregate([{$limit: 8},{$skip: 2}])`
   d. `db.emp_detail.aggregate([{$skip: 2},{$limit: 10}])`

4. Which of the following aggregation pipeline stages in MongoDB provides you to write the results of the aggregation pipeline to a collection?

   a. `$project`
   b. `$group`
   c. `$out`
   d. `$merge`

5. Which of the following string expression operator is used to split a string into substrings based on a delimiter and returns an array of substrings?

   a. `$substr`
   b. `$split`
   c. `$trim`
   d. `$substrBytes`

Answers to Test Your Knowledge

1	a, b, c
2	c
3	a, b
4	c, d
5	b

Try It Yourself

1. Create a database named Inventory and a collection named `sales_invent`.
2. Insert the following four documents into the `sales_invent` collection.

```
[
   {
     customername: "Richard",
     gender:"M",
     purchased_product:"cereals",
     quantity:6,
     price:60
     },
   { customername: "Williams",
     gender:"M",
     purchased_product:"Vegetables",
     quantity:10,
     price:150
  },
   { customername: "Emma",
     gender:"F",
     purchased_product:"Fruits",
     quantity:8,
     price:200
   },
   { customername: "John",
     gender:"M",
     purchased_product:"Baby Food",
     quantity:3,
     price:300
  }
 { customername: "Smith",
     gender:"M",
     purchased_product:"Fruits",
     quantity:5,
     price:180
  }
]
```

Using the `sales_invent` collection, perform the following tasks:

3. Exclude the `_id` field and display only the first ten documents of the `sales_invent` collection which include only the fields: `customername`, `purchased_product`, and `price`.
4. Use aggregation pipeline stages to group the documents by the `purchased_product` field and calculate the `Total sale` amount per product. Return only the products with `Total sale` amount is greater than or equal to 500.
5. Use aggregation pipeline stages to display only the first three documents of the `sales_invent` collection which includes only the `customername` and `purchased_product` fields where the `customername` is arranged in ascending order.
6. Add a field named `product_type:edibles` to all documents in the `sales_invent` collection.
7. In the first stage of aggregation pipeline, group by the `purchased_product` field and add the `quantity` fields into a new field named `Total_quantity`. In the second stage, write the output of first stage documents to a `Product_report` collection in the same `Inventory` database.
8. Use the arithmetic expression operator to calculate the `total price` calculated as `(price*quantity)` only for the last three documents.
9. Display only the product details where the `quantity` is greater than `5`.
10. Use the string expression operator to concatenate the `customername` and the `purchased_product` field as `cutomername – purchased_product` in a new field named `customer_detail`. Display the details only for the `male` customer(s).