6 Getting acquainted with MongoDB – Part 2

6.1 Source

This text is fully based on https://docs.mongodb.com/manual/aggregation/

6.2 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 (deprecated, not in this course), and single purpose aggregation methods.

The examples in this document use the zipcodes collection. This collection is available at: media.mongodb.org/zips.json.

- Create a new database called info
- Add a new collection zips to info
- Go to media.mongodb.org/zips.json. Select + Copy everything (Ctrl + A / Ctrl + C)
- Right click on zips and choose Import data
- Select Format JSON and click on the button Configure
- Click on the button 'Paste from Clipboard'
- Click on the button Run

Each document in the zipcodes collections has the following form:

```
{
    "_id" : "12838",
    "city" : "HARTFORD",
    "loc" : [
        -73.404946,
        43.349281
    ],
    "pop" : NumberInt(679),
    "state" : "NY"
}
```

- The _id field holds the zip code as a string.
- The city field holds the city.
- The loc field holds the location as a latitude longitude pair.
- The pop field holds the population.
- The state field holds the two letter state abbreviation.

6.3 Single Purpose Aggregation Operations

Aggregation refers to a broad class of data manipulation operations that compute a result based on an input and a specific procedure. MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

Count

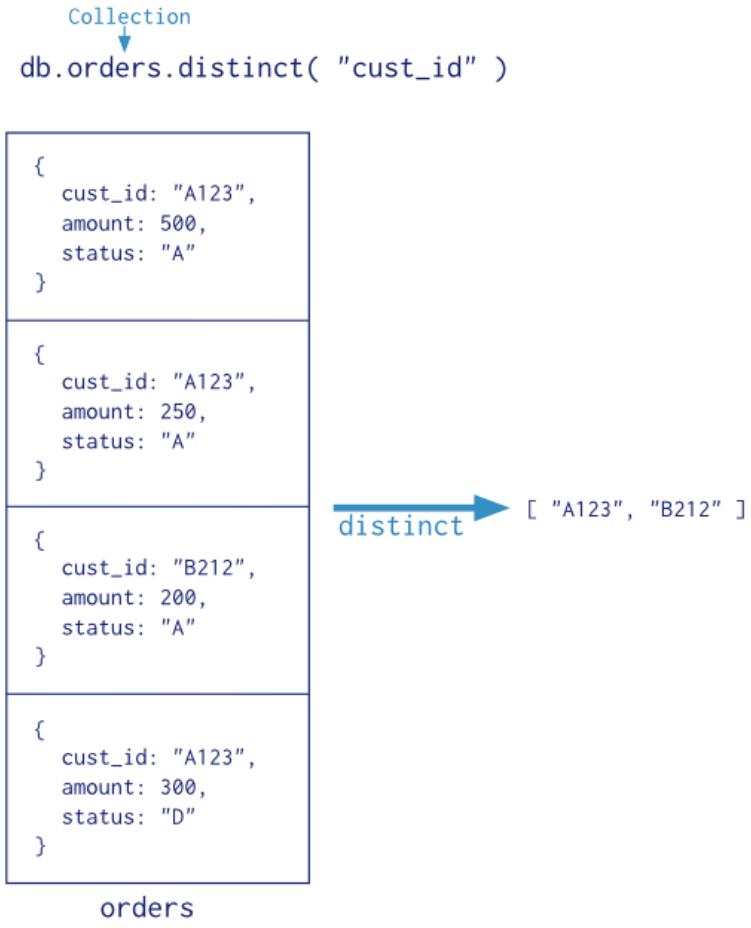
MongoDB can return a count of the number of documents that match a query.

For example: count the number of cities in the state OR

```
> db.zips.countDocuments({state : "OR"})
```

Distinct

The distinct operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents. Consider the following examples of a distinct operation:



For example: return a list of states in which each state appears only once

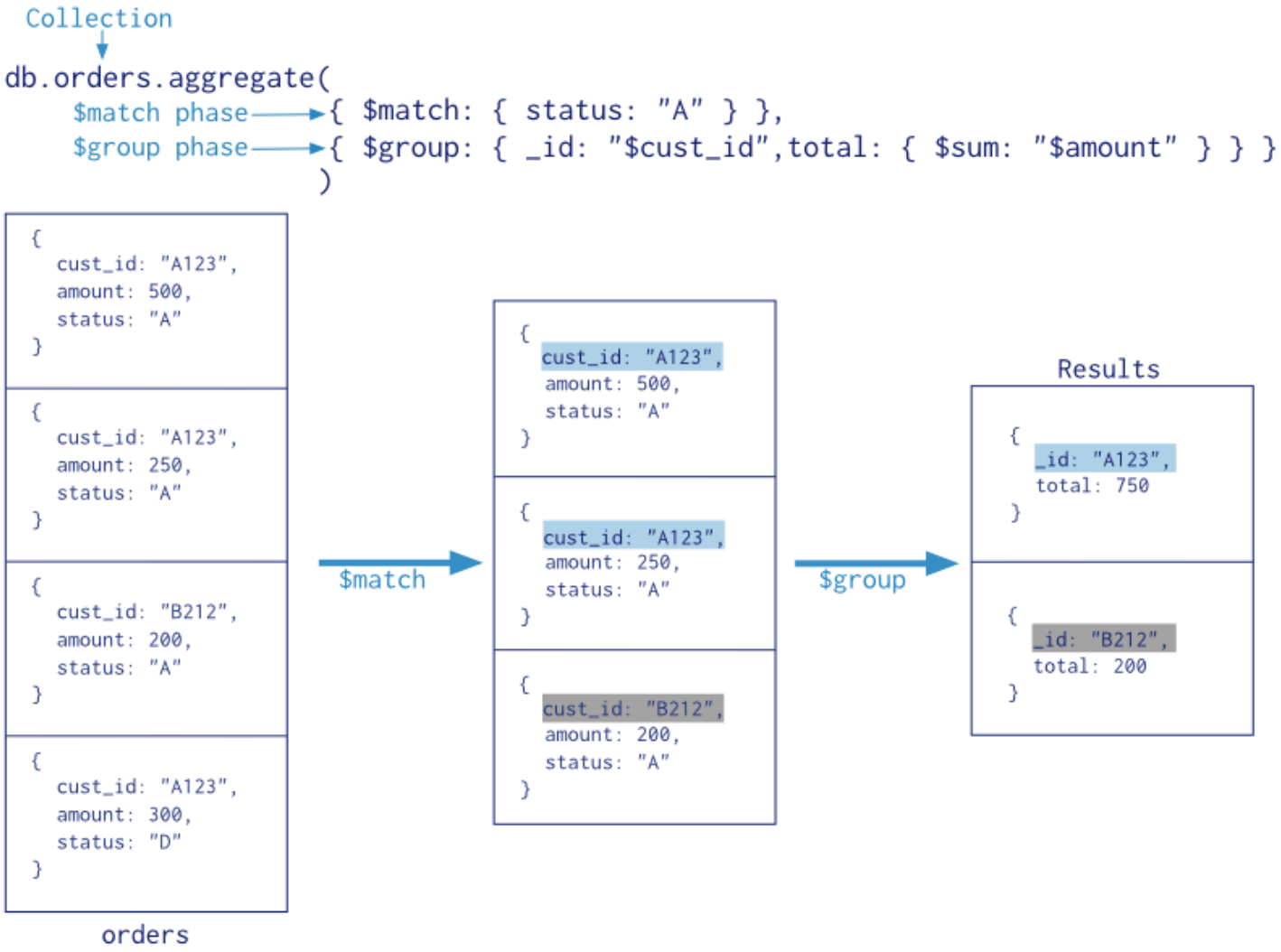
```
> db.zips.distinct("state")
```

For example: count the number of different states

```
> db.zips.distinct("state").length
```

6.4 Aggregation Pipeline

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. 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.

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.

MongoDB provides the db.collection.aggregate() method to run the aggregation pipeline.

Starting in MongoDB 4.2, you can use the aggregation pipeline also for updates.

6.5 Examples Zipcodes

6.5.1 Example 1 Return States with Populations above 10 Million

The following aggregation operation returns all states with total population greater than 10 million:

```
db.zips.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

In this example, the aggregation pipeline consists of the $group stage followed by the $match stage:

- 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.
  The new per-state documents have two fields: the _id field and the totalPop field. The _id field contains the value of the state; i.e. the group by field. The totalPop field is a calculated field that contains the total population of each state. To calculate the value, $group uses the $sum operator to add the population field (pop) for each state.
  After the $group stage, the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```
- The $match stage filters these grouped documents to output only those documents whose totalPop value is greater than or equal to 10 million. The $match stage does not alter the matching documents but outputs the matching documents unmodified.

6.5.2 Example 2 Return Average City Population by State

The following aggregation operation returns the average populations for cities in each state:

```
db.zips.aggregate( [
   { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum:
"$pop" } } },
   { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
] )
```

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

- 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.
After this stage in the pipeline, the documents resemble the following:

```
{
  "_id" : {
    "state" : "CO",
    "city" : "EDGEWATER"
  },
  "pop" : 13154
}
```

- A 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.

The documents that result from this aggregation operation resembles the following:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
}
```

6.5.3 Example 3 Return Largest and Smallest Cities by State

The following aggregation operation returns the smallest and largest cities by population for each state:

```
db.zips.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" }
```

```
    }
  }
] )
```

In this example, the aggregation pipeline consists of a $group stage, a $sort stage, another $group stage, and a $project stage:

- 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.
  At this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```
- 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.
- The next $group stage groups the now-sorted documents by the _id.state field (i.e. the state field inside the _id document) and outputs a document for each state.
  The stage also calculates the following four fields for each state. Using the $last expression, the $group operator creates the biggestCity and biggestPop fields that store the city with the largest population and that population. Using the $first expression, the $group operator creates the smallestCity and smallestPop fields that store the city with the smallest population and that population.
  The documents, at this stage in the pipeline, resemble the following:

  ```
  {
    "_id" : "WA",
    "biggestCity" : "SEATTLE",
    "biggestPop" : 520096,
    "smallestCity" : "BENGE",
    "smallestPop" : 2
  }
  ```
- The final $project stage renames the _id field to state and moves the biggestCity, biggestPop, smallestCity, and smallestPop into biggestCity and smallestCity embedded documents.

The output documents of this aggregation operation resemble the following:

```
{
  "state" : "RI",
  "biggestCity" : {
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
```

```
    "pop" : 45
  }
}
```

6.5.4 Example 4 Return the number of cities for each state

1. Project the state and city out of each document
2. Group the states by name, counting the number of occurrences
3. Sort the states by the occurrence ount, descending
4. Limit results to the first five

Each of these steps maps to a aggregation framework operator

1. `{$project: {state : 1, city : 1}}`
   This projects the state and city in each document. The syntax is similar to the field selector used in querying: you can select fields to project by specifying "fieldname" : 1 or exclude fields with "fieldname" : 0.
   After this operation, each document in the results looks like: {"_id" : id, "state" : "stateName"}. These resulting documents only exists in memory and are not written to disk anywhere.
2. `{$group : {_id : "$state", count : {$sum : 1}}}`
   This groups the authors by name and increments "count" for each document an author appears in. First, we specify the field we want to group by, which is "state". This is indicated by the _id : "$state" field. You can picture this as: after the group there will be one result document per state, so "state" becomes the unique identifier (_id).
   The second field means to add 1 to a "count" field for each document in the group.
   Note that the incoming documents do not have a "count" field; this is a new field created by the "$group".
   At the end of this step, each document in the results looks like: {"_id" : "stateName", "count" : articleCount}.
3. `{$sort : {count : -1}}`
   This reorders the result documents by the "count" field from greatest to least.
4. `{$limit : 5}`
   This limits the result set to the first five result documents.

```
db.zips.aggregate({$project : {state : 1, city : 1}},
{$group : {_id : "$state", count : {$sum : 1}}},
{$sort : {count : -1}},
{$limit : 5})
```

6.6 Examples User Preference Data

Consider a hypothetical sports club with a database that contains a users collection that tracks the user's join dates, sport preferences, and stores these data in documents that resemble the following:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
{
```

```
  _id : "joe",
  joined : ISODate("2012-07-02"),
  likes : ["tennis", "golf", "swimming"]
}
```

This collection is available as a notepadfile users.js on Chamilo.

- Add a new collection users to info
- Open IntelliShell and copy the insert-statements from the file to the shell
- Execute the insert statements

6.6.1 Example 1 Normalize and Sort Documents

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the users collection. You might do this to normalize user names for processing.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

All documents from the users collection pass through the pipeline, which consists of the following operations:

- The $project operator:
  - creates a new field called name.
  - converts the value of the _id to upper case, with the $toUpper operator. Then the $project creates a new field, named name to hold this value.
  - suppresses the id field. $project will pass the _id field by default, unless explicitly suppressed.
- The $sort operator orders the results by the name field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

6.6.2 Example 2 Return Usernames Ordered by Join Month

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

```
db.users.aggregate(
```

```
  [
    { $project :
       {
         month_joined : { $month : "$joined" },
         name : "$_id",
         _id : 0
       }
    },
    { $sort : { month_joined : 1 } }
  ]
)
```

The pipeline passes all documents in the users collection through the following operations:

- The $project operator:
  - Creates two new fields: month_joined and name.
  - Suppresses the id from the results. The aggregate() method includes the _id, unless explicitly suppressed.
- The $month operator converts the values of the joined field to integer representations of the month. Then the $project operator assigns those values to the month_joined field.
- The $sort operator sorts the results by the month_joined field.

The operation returns results that resemble the following:

```
{
  "month_joined" : 1,
  "name" : "ruth"
},
{
  "month_joined" : 1,
  "name" : "harold"
},
{
  "month_joined" : 1,
  "name" : "kate"
}
{
  "month_joined" : 9,
  "name" : "jill"
}
```

6.6.3 Example 3 Return Total Number of Joins per Month

The following operation shows how many people joined each month of the year. You might use this aggregated data for recruiting and marketing strategies.

```
db.users.aggregate(
```

```
  [
    { $project : { month_joined : { $month : "$joined" } } } ,
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum :
1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

The pipeline passes all documents in the users collection through the following operations:

- The $project operator creates a new field called month_joined.
- The $month operator converts the values of the joined field to integer representations of the month. Then the $project operator assigns the values to the month_joined field.
- The $group operator collects all documents with a given month_joined value and counts how many documents there are for that value. Specifically, for each unique value, $group creates a new "per-month" document with two fields:
    - _id, which contains a nested document with the month_joined field and its value.
    - number, which is a generated field. The $sum operator increments this field by 1 for every document containing the given month_joined value.
- The $sort operator sorts the documents created by $group according to the contents of the month_joined field.

The operation returns results that resemble the following:

```
{
  "_id" : {
    "month_joined" : 1
  },
  "number" : 3
},
{
  "_id" : {
    "month_joined" : 2
  },
  "number" : 1
}, …
```

6.6.4 Example 4 Return the Five Most Common "Likes"

The following aggregation collects top five most "liked" activities in the data set. This type of analysis could help inform planning and future development.

```
db.users.aggregate(
  [
    { $unwind : "$likes" },
    { $group : { _id : "$likes" , number : { $sum : 1 } } },
```

```
    { $sort : { number : -1 } },
    { $limit : 5 }
  ]
)
```

The pipeline begins with all documents in the users collection, and passes these documents through the following operations:

- The $unwind operator separates each value in the likes array, and creates a new version of the source document for every element in the array.

  EXAMPLE

  Given the following document from the users collection:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : ["golf", "racquetball"]
  }
  ```

  The $unwind operator would create the following documents:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : "golf"
  }
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : "racquetball"
  }
  ```

- The $group operator collects all documents with the same value for the likes field and counts each grouping. With this information, $group creates a new document with two fields:
  - _id, which contains the likes value.
  - number, which is a generated field. The $sum operator increments this field by 1 for every document containing the given likes value.
- The $sort operator sorts these documents by the number field in reverse order.
- The $limit operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

```
{
  "_id" : "golf",
```

```
  "number" : 5
},
{
  "_id" : "tennis",
  "number" : 3
},
{
  "_id" : "racquetball",
  "number" : 3
},
{
  "_id" : "darts",
  "number" : 2
},
{
  "_id" : "basketball",
  "number" : 1
}
```

6.7 Pipeline Operations

Each operator receives a stream of documents, does some type of transformation on these documents, and then passes on the results of the transformation. If it is the last pipeline operator, these results are returned to the client. Otherwise, the results are streamed to the next operator as input.

Operators can be combined in any order and repeated as many times as necessary. For example, you could "$match", "$group", and then "$match" again with different criteria.

`$match`

`$match` filters documents so that you can run an aggregation on a subset of documents. For example, if you only want to find out stats about states in Oregon, you might add a "$match" expression such as

```
db.zips.aggregate({$match : {state : "NY"}})
```

"$match" can use all of the usual query operators ("$gt", "$lt", "$in", etc.). Generally, good practice is to put "$match" expressions as early as possible in the pipeline, because it filters out unneeded documents quickly.

`$project`

Projection is much more powerful in the pipeline than it is in the “normal” query language.

"$project" allows you to extract fields from subdocuments, rename fields, and perform interesting operations on them. The simplest operation "$project" can perform is simply selecting fields from your incoming documents. To include or exclude a field, use the same syntax you would in the second argument of a query. The following would return a result document containing one field, "state", for each document in the original collection:

```
db.zips.aggregate({$match : {"state" : "NY"}},
{$project : {state : 1, _id : 0}})
```

By default, "_id" is always returned if it exists in the incoming document (some pipeline operators remove the "_id" or it can be removed in a former projection). You can exclude it as above. Inclusion and exclusion rules in general work the same way that they do for “normal” queries.

You can also rename the projected field. For example, if you wanted to return the "_id" of each user as "stateId", you could do:

```
> db.zips.aggregate({$match : {"state" : "NY"}},
{$project : {stateId : "$_id", _id : 0}})
```

The "$fieldname" syntax is used to refer to fieldname’s value in the aggregation framework. Thus, "$_id" is replaced by the "_id" field of each document coming through the pipeline.

Note that you must specifically exclude "_id" to prevent it from returning the field twice, once labeled "stateId" and once labelled "_id". You can use this technique to make multiple copies of a field for later use in a "$group", say.

Mathematical expressions.

Arithmetic expressions let you manipulate numeric values. You generally use these expressions by specifying an array of numbers to operate on. For example, the following expression would multiply the "population" by 2 for each city of the state Oregon

```
db.zips.aggregate({$match : {"state" : "NY"}},
{$project: {twicePop: {$multiply : ["$pop",2]}}})
```

The following expression would divide the "population" by 2 for each city of the state Oregon

```
db.zips.aggregate({$match : {"state" : "NY"}},
{$project: {halvePop: {$divide : ["$pop",2]}}})
```

The plus – operator is $add en the minus-operator is $subtract

Date expressions

Many aggregations are time-based: What was happening last week? Last month? Over the last year? Thus, aggregation has a set of expressions that can be used to extract date information in more useful ways: `"$year"`, `"$month"`, `"$week"`, `"$dayOfMonth"`, `"$dayOfWeek"`, `"$dayOfYear"`, `"$hour"`, `"$minute"`, and `"$second"`. You can only use date operations on fields stored with the date type, not numeric types.

String expressions

There are a few basic string operations you can do as well. Their signatures are:

```
"$substr" : [expr, startOffset, numToReturn]
```

This returns a substring of the first argument, starting at the startOffset-th byte and including the next numToReturn bytes (note that this is measured in bytes, not characters, so multibytes encodings will have to be careful of this). expr must evaluate to a string.

`"$concat" : [expr1[, expr2, ..., exprN]]`
Concatenates each string expression (or string) given.

`"$toLower" : expr`
Returns the string in lower case. expr must evaluate to a string.

`"$toUpper" : expr`
Returns the string in upper case. expr must evaluate to a string.

Case-affecting operations are only guaranteed to work on characters from the Roman alphabet.

For example

Produce a list of city state for each city in the state NY

```
db.zips.aggregate({$match : {"state" : "NY"}},
{$project: {city_state : {$concat : ["$city"," ","$state"]}}})
```

$group

Grouping allows you to group documents based on certain fields and combine their values. If we want to know the population for each state, we would group by the "state" field.

When you choose a field or fields to group by, you pass it to the "$group" function as the group's "_id" field.

Grouping operators

`"$sum" : value`
This adds value for each document and returns the result.

`"$avg" : value`
Returns an average of all input values seen during the group.

`"$max" : expr`
Returns the greatest value of any of the inputs.

`"$min" : expr`
Returns the smallest value of any of the inputs.

`"$first" : expr`
This returns the first value seen by group, ignoring subsequent values. This is only sensible to use when you know the order that the data is being processed in: that is, after a sort.

`"$last" : expr`
This is the opposite of the previous; it returns the last value seen by the group.

"$max" and "$min" look through each document and find the extreme values. Thus, these operators work well when you do not have sorted data and are a bit wasteful when data is sorted.

For example

Calculate the number of cities and the total population for each state. Return the 5 states with the largest population

```
db.zips.aggregate( {$group : {_id : "$state", count : {$sum:1}, totpop : {$sum : "$pop"}}}, {$sort : {totpop : -1}},  {$limit : 5})
```

What is the maximum population for 1city for each state? Return the 5 states with the largest cities

```
db.zips.aggregate( {$group : {_id : "$state", maxpop : {$max:"$pop"}}}, {$sort : {maxpop : 1}},  {$limit : 5})
```

$ unwind

Unwinding turns each field of an array into a separate document. For example, if we had a blog with comments, we could use unwind to turn each comment into its own "document":

```
db.blog.insert({"author" : "lisa", "post" : "Hello, world!", "comments" : [{"author" : "mark", "text" : "Nice post" }, {"author" : "bill", "text" : "I agree"}]})
```

```
db.blog.aggregate({$unwind:"$comments"})
```

This is particularly useful if you want to return certain subdocuments from a query: "$unwind" the subdocuments and then "$match" the ones you want. For example, it is impossible in the normal query language to return all comments by a certain user and only those comments, not the posts they commented on. However, by projecting, unwinding, and matching, it becomes trivial:

```
db.blog.aggregate({$project : {comments : "$comments"}}, {$unwind : "$comments"}, {$match : {"comments.author" : "mark"}})
```

$sort

You can sort by any field or fields, using the same syntax you would with the "normal" query language. Possible sorts are 1 (for ascending) and −1 (for descending).

$limit

$limit takes a number, n, and returns the first n resulting documents.

For example

Return a list of the 5 first cities ordered from z to a with their states

```
db.zips.aggregate({$project: {state : "$state",city : "$city"}},{$sort : {city:1}},{$limit : 5})
```

$skip

$skip takes a number, n, and discards the first n documents from the result set. As with "normal" querying, it isn't efficient for large skips, as it must find all of the matches that must be skipped and then discard them.

For example

Return a list of the cities 100 .. 110 ordered from z to a with their states

```
db.zips.aggregate({$project: {state : "$state",city : "$city"}},{$sort : {city:1}},{$skip:100},{$limit : 10})
```

6.8 Exercises

6.8.1 Exercise 1

The collection cheeses is available as a notepadfile on Chamilo.

- Add a new collection cheeses to info
- Open IntelliShell and copy the insert-statements from the file to the shell
- Execute the insert statements

Specify the number of cheeses per cheesetype. Put the cheesetype in capital letters.

Give the number of cheeses per cheese factory with at least 2 variants.

6.8.2 Exercise 2

The collection bel20 is available as a notepadfile on Chamilo.

- Add a new collection bel20 to info
- Open IntelliShell and copy the insert-statements from the file to the shell
- Execute the insert statements

Give per share the average closing price, the minimum closing price, the maximum closing price and the average number of shares traded per day.

Give per share the minimum closing price and the week in which this minimum closing price occurred. E.g.

KBC - minprice = 39.1 - week = 44

Elia - minprice = 33.38 - week = 46