VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

MongoDB Aggregation

Lecturer* : *Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Plan for MongoDB Aggregation

- Single Purpose Aggregation
- Aggregation Pipeline
- Map – Reduce

  – ***Reedings:***
    - *Have a look at Readings on the Home Page*

Aggregation Operations in MongoDB

- Three classes of aggregation commands:
  - Aggregation Pipeline,
  - Map – Reduce, and
  - Single Purpose Aggregation
- All three commands process documents from a single collection
- **Aggregation Pipeline** uses native MongoDB operations and is the preferred aggregation method
- **Map – Reduce:**
  - Has two phases: a map phase that processes each input document and emits objects, and a reduce phase that combines the output of the map operation
  - It is based on custom JavaScript functions and aimed for very large collection

Single Purpose Aggregation (SPA)

- **Single Purpose Aggregation** is simple but limited in scope, everything supported by it can be done by Aggregation Pipeline
- In principle, SPA supports: count, distinct, and group operations

```
db.myclasses.count()
```

  - Returns the number of all documents in `myclasses`

```
db.myclasses.count({
'course.code': {$regex:/^SWEN/}})
```

  - Returns the number of SWEN documents in `myclasses`

```
db.myclasses.distinct('course.code',
{coordinator: "Pavle"})
```

  - Returns the distinct course codes where `coordinator` is `"Pavle"`

Aggregation Pipeline

- Aggregation pipeline consists of *stages*
- Each stage transforms the documents as they pass through the pipeline:
  - Some stages transform old into new documents,
  - Others filter out documents,
  - The same stage can appear multiple times in a pipeline
- For the aggregation pipeline, MongoDB provides the

```
db.collection.aggregate()
```

method in the mongo shell

- Pipeline stages appear in an array
- Documents pass through the stages in sequence

```
db.collection.aggregate([{<stage>},...])
```

Stage Operators (1)

- **`$group`**:
  - Groups all input documents by a specified identifier expression,
  - Applies the accumulator expression(s), if specified, to each group,
  - Outputs one document per each distinct group,
  - The output documents only contain the `_id` field and, if specified, accumulated fields
- **`$limit`**:
  - Passes only the first *n* documents unmodified to the pipeline where *n* is the specified limit
- **`$match`**:
  - Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage,
  - Uses standard MongoDB query selector documents

Stage Operators (2)

- **`$out`**:
  - Writes the resulting documents of the aggregation pipeline to a collection
  - To be used, must be the last stage in the pipeline
- **`$project`**:
  - Reshapes each document in the stream by adding new or removing existing fields,
  - For each input document, outputs one document
- **`$redact`**:
  - Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves,
  - Incorporates the functionality of `$project` and `$match`,
  - For each input document, outputs at most one document

Stage Operators (3)

- **`$skip`**:
  - Skips the first *n* documents and passes the remaining documents unmodified to the pipeline
- **`$sort:`**
  - Reorders the document stream by a specified sort key(s)
  - For each input document, outputs one unmodified document
- **`$unwind`**:
  - Deconstructs an array field from the input documents to output a document for *each* array element
  - Each output document replaces the array with an element value
  - For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array

A Document Example

```
{
_id: ObjectId(“33667997ab01”)
  course: {code: “SWEN432”,
           title: “Advanced DB”},
  year: 2014,
  coordinator: “Pavle”,
  no_of_st: 11,
  prerequisites: [“SWEN304”, “COMP261”,
  “NWEN304"],
rating: 1.5,
last_modified: ISODate("2014-02-
  13T02:14:37.948Z")
}
```

Pipeline Aggregation Example (1)

- Designate the popularity of 2014 four hundred level courses having more than 14 students as “favoured” and those having less than 15 students as “less favoured”
  - At the start we need to filter our collection and to retain only four hundred level courses
  - So, the first stage operator is going to be `$match`

```
db.myclasses.aggregate([{
$match: … }, …
 ] )
```

Pipeline Expressions (1)

- Some pipeline stages take pipeline expressions as their operands
- Pipeline expressions specify the transformation to apply to the input documents
- Expressions have a *document* structure and can contain other *expressions*
- Pipeline expressions can only operate on the current document in the pipeline and cannot refer to data from other documents:
  - Expression operations provide in-memory transformation of documents

Pipeline Expressions (2)

- All expressions, except accumulator expression are stateless and are only evaluated when seen by the aggregation process
- The accumulators, used with the `$group` pipeline operator, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline
- Generally, expression operators take an array of arguments and have the following general form:

```
{<operator>: [<argument1>, ... ] }
```

- If an operator accepts a single argument, then

```
{<operator>: <argument>}
```

Expression Operators (1)

- **Boolean Operators**: **`$and, $or,`** and **`$not`**
  - In addition to the false boolean value, Boolean expressions evaluate as false the following: null, 0, and undefined values
- **Arithmetic Operators: `$add, $divide, $mod, $multiply, $subtract`**
  - Some arithmetic expression operators (**`$add`** **and** **`$subtract`**) can also support date arithmetic
- **Comparison expression operators**: **`$cmp, $eq, $gt, $gte, $lt, $lte, $ne`**
- **`$cmp:`**
  - Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and -1 if the first value is less than the second.

Expression Operators (2)

- **String Operators**: **`$concat, $strcasecmp, $substr, $toLower, $toUpper`**
- **Array Operators: `$size`**
  - Returns the number of elements in the array
- **Variable Operators**: **`$let`** and **`$map`**
- **`$let:`**
  - Defines variables for use within the scope of a subexpression and returns the result of the subexpression
- **`$map:`**
  - Applies a subexpression to each element of an array and returns the array of resulting values in order

Pipeline Aggregation Example (1) (cont.)

- Designate the popularity of 2014 four hundred level courses having more than 14 students as “favoured” and those having less than 15 students as “less favoured”
  - Now, we are able to define the filter to find 2014 four hundred courses using comparison operator `$eq` and string operator `$substr`

```
db.myclasses.aggregate([{
$match: {year: 2014, {$eq: [4, {$substr:
['course.code', 4, 1] } ] } } },…
] )
```

Set Expression Operators

- **Set Operators** perform set operation on single level arrays, treating arrays as sets
  - The operation filters out duplicates in the result
  - The order of the elements in the output array is unspecified
- Operators:
  - **`$allElementsTrue`**
  - **`$anyElementTrue`**
  - **`$setDifference`**
  - **`$setEquals`**
  - **`$setIntersection`**
  - **`$setIsSubset`**
  - **`$setUnion`**

Date Expression Operators

- The following operators return a number for a date:
  - **$dayOfMonth,**
  - **$dayOfWeek,**
  - **$dayOfYear,**
  - **$hour,**
  - **$millisecond,**
  - **$minute,**
  - **$month,**
  - **$second,**
  - **$week,**
  - **$year**

Conditional Operators

- **$cond**
  - A ternary operator that evaluates one expression, and depending on the result, returns the value of one of the other two expressions,
  - Accepts either three expressions in an ordered list or three named parameters

```
{ $cond: { if: <boolean-expression>, then: <true-case>,
else: <false-case> } }
```

- **$ifNull**
  - Returns either the non-null result of the first expression or the result of the second expression if the first expression results in a null result,
  - Null result encompasses instances of undefined values or missing fields,
  - Accepts two expressions as arguments,
  - The result of the second expression can be null

Accumulator Expression Operators (1)

- **Accumulator Operators** are available for the `$group`, and some also for `$project` stage
- Accumulators compute values by combining documents that share the same group key (`_id`)
- Group key has to be defined
- Accumulators:
  - Take as input a single expression,
  - Evaluate the expression once for each input document, and
  - Maintain their state for each group of documents

Accumulator Expression Operators (2)

- The `$group` stage has the following prototype form

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

- The `_id` field is *mandatory*; however, you can specify an `_id` value of `null` to calculate accumulated values for all the input documents as a whole
- The remaining computed fields are *optional* and if exist, are computed using the `<accumulator>` expression operators

Accumulator Expression Operators (3)

- **`$addToSet`**
  - Returns an array of *unique* expression values for each group
- **`$avg`**
- **`$first`**
  - Returns a value from the first document for each group
- **`$last`**
  - Returns a value from the last document for each group
- **`$max`**
- **`$min`**
- **`$push`**
  - Returns an array of expression values for each group
- **`$sum`**

Field Path to Access Fields

- Expressions in the `$group` and `$project` stages use ***field path*** to access ***value*** fields in the ***input documents*** by prefixing the field name with a dollar sign `$` :
  - `"$year"` to specify the field path to the value of `year`, or
  - `"$course.title"` to specify the field path to the value of `course.title`
- Since the field path is a value, it has to be quoted
- The output from a `db.collection.aggregate()` having any of `$group`, `$project`, or `$reshape` stages contains only those fields that are defined using field path, or are generated by expressions in ***each of these stages***

Pipeline Aggregation Example (1) (cont.)

- Designate the popularity of 2014 four hundred level courses having more than 14 students as “favoured” and those having less than 15 students as “less favoured”
  - Now, we proceed to the next stage that is `$project` and
  - Use field path to rename some field names and use the `$cond` expression operator to decide on values of a newly introduced `popularity` field

```
db.myclasses.aggregate([{
$match: {year: 2014, {$eq: [4, {$substr:
['course.code', 4, 1] } ] } } }, {
$project: {_id: null, code: '$course.code',
title: '$course.title', popularity:
{$cond: {if: {$gt: ['$no_of_st', 14] },
then: 'favoured', else: 'less favoured' } }
} } ] )
```

Pipeline Aggregation Examples (2)

- Display course `code`, `title`, `year`, and `rating` for all courses. If a course does not have a `rating` field, or it is `null`, display `rating: Unspecified`

```
db.myclasses.aggregate([{
$match: {'course.code': {$exists: true}} },
{
$project: {_id:0, code: '$course.code',
title: '$course.title', year: '$year',
rating: { $ifNull: [ "$rating",
"Unspecified" ] } } }
] )
```

Pipeline Aggregate Examples (3)

- Find the number of all students enrolled in `year: 2015` courses

```
db.myclasses.aggregate([{
$match: {year: 2015} }, {
$group: {_id: null,
total_enrollment: {$sum: "$no_of_st" }}}
])
```

Pipeline Aggregate Examples (4)

- Retrieve course codes by `year` and month
  - Take month from `last_modified`

```
db.users.aggregate( [
{$match: {last_modified: {$exists: true}}},
{$project: {year: “$year”,
      month: {$month: "$last_modified"},
      code: "$course.code", _id: 0}},
{$sort: {year: 1, month: 1} } ]  )
```

- The field `last_modified` has to exist in all documents that enter the `$project` stage

Pipeline Aggregate Examples (5)

- Find the number of courses per group (like: SWEN, NWEN, COMP, ECEN, ENGR)
  - Display in a descending order of `number` values

```
db.myclasses.aggregate([
{$group:
{_id : "$course.code", number: {$sum : 1}
}},
{$sort: {"number" : -1} }
])
```

- The aggregate expression operator `$sum` counts courses by adding 1 to the current number

Pipeline Aggregate Examples (6)

- Find the five courses that appear most often as prerequisites
  - Display in descending order

```
db.myclasses.aggregate([
{$unwind: "$prerequisites"},
{$group: {_id:"$prerequisites", number:
{$sum : 1}}},
{$sort: {number: -1}},
{$limit: 5} ] )
```

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

Pipeline Aggregate Examples (7)

- Find groups (like: SWEN, NWEN, COMP, ECEN,...) having at least 500 enrolments in 2015
  - Display in descending order of enrolment numbers

```
db.myclasses.aggregate(
{$match: {year: 2015}},
{$group:
{_id: {$substr: ["$course.code", 0, 4]},
enrolled: { $sum : "$no_of_st"} } },
{$match: {enrolled: {$gte: 500} } } )
```

- The `$substr` expression operator extracts the first four characters of the `course.code` field as the group identifier value

Pipeline Aggregate Examples (8)

- Return course codes of classes having the smallest and the greatest number of students enrolled within each group (like: SWEN, NWEN, COMP, ECEN,...) in 2015

```
db.myclasses.aggregate({$match:
{year: 2015, no_of_st: {$exists: true}}},
{$sort: {no_of_st: -1} },
{$group: {_id : {$substr: ["$course.code", 0,
4]}},
biggestGroup: {$first: "$course.code"},
bigestEnrollment: {$first: "$no_of_st"},
smallestGroup: {$last: "$course.code"},
smallestEnrollment: {$last: "$no_of_st"} } }
)
```

Summary

- Three ways to do aggregation:
  - Pipeline (preferred),
  - Map Reduce, and
  - Single purpose
- Aggregation Pipeline consists of stages
- Stage operators:
  - `$match, $group, $project, $sort, $skip`,...
- Each stage operator contains expression(s) that specify transformations of documents
- Expression operators:
  - Boolean, Arithmetic, Comparison, String, Set, Date, Accumulator,...
- Expressions in the `$group` and `$project` stages use field path to access fields in the input documents
- Examples show use cases