COMP5338 – Advanced Data Models

Week 3: MongoDB – Aggregation Framework

Dr. Ying Zhou
School of Information Technologies

Outline

- Review
- Aggregation
  - Pipeline stages
  - Operators

Review

- Document Storage Systems store data as *semi-structured* document: XML or JSON as two dominant *semi-structured* formats
  - Semi-structured data is self-describing
- MongoDB is a popular document storage system that stores data as Binary representation of JSON document (BSON)
- Documents with sim g a particular type of entity are stored in t
- A database is used representing related entities
- All CRUD operations (find, update, ins) target single collection
- Query criteria, projection and modifier are expressed as JSON document

Null, empty string and related operators

- Null (or null) is a special data type
  - Similar to **`None, Null`** or **`Nil`** in any programming language
  - It has a singleton value expressed as **`null`**
  - Indicating no value is given
- The interpretation of null is different depending on where it appears
- It might represe
  - The field exists, but has no value
  - The field does not exits
- This is different to given a field an empty string “” as value

Null query example

- Collection revisions document sample

```
{   "_id" : ObjectId("5799843ee2cbe65d76ed919b"),
    "title" : "Hillary_Clinton",
    "timestamp" : "2016-07-23T02:02:06Z",
    "revid" : 731113635,
    "user" : "BD2412",
    "parentid" : 7311135
    "size" : 251742,
    "minor" : ""}
```

- We need a field to indicate if a revision is minor or not. The original schema uses a field with empty string value to indicate a minor revision; a document without this field would be a non-minor revision.

https://docs.mongodb.com/manual/tutorial/query-for-null-fields/

Querying for null or field existance

- Queries
  - `db.revisions.find({minor:{$exists:true}})`
    - Find all documents that a field called **`minor`** exists
  - `db.revisions.find(`where the `{minor:""})`
    - Find all documents whose **`minor`** field has a value of “”, empty string
  - `db.revisions.`
    - Find all docu **`or`** field or the value of **`minor`** field is null
  - `db.revisions.find({minor:{$ }})`
    - Find all documents that does not have a field called **`minor`**

It is possible to set the value to `null`

```
db.revisions.insertOne({title:"nulltest",
   "timestamp" : "2018-08-14T02:02:06Z",
    "revid" : NumberLong(7201808141159),
    "user" : "BD2412",
    "parentid" : 731113573,
    "size" : NumberInt(251900),
    "minor":null})
db.revisions.insertOn
    "timestamp" : "2018-08-14T02:02:
    "revid" : NumberLong(201808141157
    "user" : "BD2412",
    "parentid" : NumberLong(731113573),
    "size" : NumberInt(251800)})
```

`db.revisions.find({minor:null})` would return both documents

`db.revisions.find({minor:{$exists:true}})` can differentiate the two

Aggregation

- Simple and relatively standard data analytics can be achieved through **aggregation**
  - Grouping, summing up value, counting, sorting, etc
  - Running on the DB engine instead of application layer

- Several options
  - Aggregation Pip
  - MapReduce
    - Through JavaScript Functions
    - Is able to do customized aggregations

Aggregation Pipeline

- Aggregation pipeline consists of multiple stages
  - Stages are specified using **pipeline operators** such as **$match, $group,$project, $sort** and so on
    - This is similar to SQL's WHERE, GROUP BY, SORT BY etc
    - Each stage is expressed as an object enclosed by curly bracket
  - Various **expressions** can be specified in each stage
    - To filter docu ulation on an document
      - `$substr`, `$`
  - **$group** stage can specify **accum** rform calculation on documents with the same group k

```
db.collection.aggregate( [
        { pipeline operator: {expression/accumulator,…, expression/accumulator} },
        { pipeline operator: {expression/accumulator,…, expression/accumulator} },
        ...
        ] )
```

Aggregation Example

```
select  cust_id as _id, SUM(amount) as total
        from orders
        where  status = “A”
        group by cust_id
```

Typical aggregation stages

- $match
- $group
- $project
- $sort
- $skip
- $limit
- $count
- $sample
- $out
- $unwind
- $lookup

$match stage

- **`$match`**
  - Filter the incoming documents based on given conditions
  - Format:

```
{$match: {<query>}}
```

    - The query document is the same as those in the **`find`** query
  - Example:

```
db.revisions.ag                                    :{$lt: 250000 }}}])
```

Has the same effect as

```
db.revisions.find({size :{$lt: 250000 }})
```

$group stage

- $group
  - Groups incoming documents by some specified expression and outputs to the next stage a document for each distinct group
    - The **`_id`** field of the output document has the value of the group key for each group

    - The stage can

```
{ $group: {_id:
           <field1>:{accumulato              ion>},
           …},
```

  - To specify the whole collection as a group, give **`_id`** field **`null`** value
  - Use *field path* to access fields in the document and set one or many as the value of the **`_id`** field
    - "**`$title`**", or "**`$address.street`**"
  - There are predefined accumulators: `$sum`, `$avg`, `$first`, `$last`, etc

$group stage example

- Find the earliest revision time in the whole collection

```
db.revisions.find({}, {timestamp:1, _id:0})
        .sort({timestamp:1})
        .limit(1)
```

Accumulator: field path

```
db.revisions.aggregate([
  {$group: {_id:nu                        imestamp"}}}
])
```

- Find the earliest revision time   e in the collection

```
db.revisions.aggregate([
  {$group: {_id:"$title", earliest: {$min: "$timestamp"}}}
])
```

field path

$group stage example (cont'd)

- Find the number of revisions made on each page by each individual user
  - This would require grouping based on two fields: title and user
  - We need to specify these two as the _id field of the output document

Composite type as _id

```
db.revisions.aggreg
  {$group: {_id:{ti                                  er"},
                          rev_count: {$sum: 1}
])
```

$group by more than one field

```
{_id:ObjectId("…"), title: "DT", user:"A", size:123, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"B", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"B", size:125, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"A", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"A", size:125, timestamp:…, … }
```

```
e:"$title",user:"$user"},
v_count: {$sum: 1}}}
```

```
{_id: {title: "DT", user:"A"}
{_id: {title: "HC", user:"B"}, rev_count: 1}
{_id: {title: "DT", user:"B"}, rev_count: 1}
{_id: {title: "HC", user:"A"}, rev_count: 1}
```

$group examples (cont'd)

- Accumulators do not just return a single value, we can use accumulators to create an array to hold data from incoming documents
- What do the following two commands do:

```
db.revisions.aggregate([
 {$group: {_id:"$titl
            revs: {$pu                         :"$timestamp"}}}
}])
```

```
db.revisions.aggregate([
 {$group: {_id:"$title",rev_users: {$addToSet:"$user"}}}
])
```

$push accumulator

```
{_id:ObjectId("…"), title: "DT", user:"A", size:123, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"B", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"B", size:125, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"A", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"A", size:125, timestamp:…, … }
```

```
db.revisions.aggregate([

"$user",timestamp:"$timestamp"}}}
}])
```

```
{ _id: "DT",
  revs:[
         {user:"A",timestamp:…},
         {user:"B",timestamp:…},
         {user:"A",timestamp:..}
     ]}
{ _id:"HC",
  revs:[
         {user:"A", timestamp:…},
         {user:"B", timestamp:…}
      ]}
```

$addToSet accumulator

```
{_id:ObjectId("…"), title: "DT", user:"A", size:123, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"B", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"B", size:125, timestamp:…, … }
{_id:ObjectId("…"), title: "HC", user:"A", size:113, timestamp:…, … }
{_id:ObjectId("…"), title: "DT", user:"A", size:125, timestamp:…, … }
```

```
ate([
tle",
:{$addToSet:"$user"}}}
])
```

```
{ _id:"DT",
  rev_users:["A", "B"]
}
{ _id:"HC",
  rev_users:["A", "B"]
}
```

$project stage

- `$project`
  - Reshape the document by including/excluding field, *adding new fields*, *resetting the value of existing field*
  - More powerful than the *project* argument in **`find`** query
  - Format

`{$project: {<sp`

  - The specification e followed by a single value indicating the inclusion or exc s

  - Or it can be a field name (existing ed by an expression to compute the value of the field

  `<field>: <expression>`

  - In the expression, existing field from incoming document can be accessed using field path: "*$fieldname*"

$project examples

- Find the age of each title in the collection, where the age is defined as the duration between the last and the first revision of that title, assuming the timestamp is of ISODate type

```
db.revisions.aggregate([
{$group: {_id:"$t
         first:
          last:  {$max:"$timest
{$project: {_id: 0,
            title: "$_id",
            age: {$subtract:["$last","$first"]}}}
])
```

$group then $project

```
{_id:ObjectId(“…”), title: “DT”, timestamp:“2016-07-01 00:03:46.000Z”, … }
{_id:ObjectId(“…”), title: “HC”, timestamp:“2016-07-01 00:55:44.000Z”, … }
{_id:ObjectId(“…”), title: “DT”, timestamp:“2016-07-15 12:22:35.000Z”, … }
{_id:ObjectId(“…”), title: “HC”, timestamp:“2016-07-28 00:03:58.000Z”,… }
{_id:ObjectId(“…”), title: “DT”, timestamp:“2016-07-28 00:20:19.000Z”, … }
```

```
{$group: {_id:"$title",
                        {$min:"$timestamp"},
                        {$max:"$timestamp"} }},
```

```
{_id:“DT”, first:“2016-07-01 00:03:46.000               07-28 00:20:19.000Z”}
{_id:“HC”, first:“2016-07-01 00:55:44.000               07-28 00:03:58.000Z”}
```

```
{$project: {_id: 0,
            title: "$_id",
            age: $subtract:["$last","$first"]}}}
```

```
{title: “DT”, age:2333793000}
{title: “HC”, age:2329694000}
```

We can combine multiple operators

```
db.revisions.aggregate([
{$group: {_id:"$title",
          first: {$min:"$timestamp"},
            last:  {$max:"$timestamp"} }},
{$project: {_id
              tit
              age:{$divide:
                  [{$subtract:["$last","$first"]},
                  86400000]}}}
              age_unit: {$literal:"day"}}}
])
```

`$sort`, `$skip`, `$limit` and `$count` stages

- **`$sort`** stage sorts the incoming documents based on specified field(s) in ascending or descending order
  - The function and format is similar to the sort modifier in **`find`** query
  - **`{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }`**
- **`$skip`** stage skips over given number of documents
  - The function and f ifier in **`find`** query
  - **`{ $skip: <posit`**
- **`$limit`** stage limits the number of do ed to the next stage
  - The function and format is similar to t er in **`find`** query
  - **`{ $limit: <positive integer> }`**
- **`$count`** stage counts the number of documents passing to this stage
  - The function and format is similar to the count modifier in find query
  - **`{ $count: <string> }`**
  - String is the name of the field representing the count

`$sample` and `$out` stages

- The **`$sample`** stage randomly selects given number of documents from the previous stage
  - **`{ $sample: { size: <positive integer> } }`**
  - Different sampling approaches depending on the location of the stage and the size of the sample and the collection
  - May fail due to
- The **`$out`** stage a given collection
  - should be the last one in the pipel
  - **`{ $out: "<output-collection>" }`**

$lookup stage

- New aggregation stages are added with major releases.
- **$lookup** stage is added since 3.2 to perform left outer join between two collections
  - The collection already in the pipeline (maybe after a few stages)
  - Another collection (could be the same one)

- For each *incoming d* the `$lookup` stage adds a new **array field** w ing documents *from the other collection*.

```
{$lookup:
     { from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from"
collection>,
       as: <output array field>
     }
}
```

`$lookup` stage (cont'd)

- The output of **`$lookup`** stage has the same number of documents as the previous stage
- Each document is augmented with an **<u>array field</u>** storing matching document(s) from the other collection
- The array could ocuments depending on th
- Missing local or foreign field is having **`null`** value

$lookup stage example

```
db.orders.aggregate([
   {
     $lookup:
       {
         from: "inventory",
         localField: "item",
         foreignFiel
         as: "invent
       }
   }
])
```

```
{"_id":1, "item":"abc", "price":12,"quantity":2 }
{"_id":2, "item":"nosku", "price":20,"quantity":1 }
{"_id":3 }
```

A document with no **item** field

orders

inventory

```
{"_id":1, "sku":"abc", description:"product 1", "instock":120}
{"_id":2, "sku":"def", description:"product 2", "instock":80 }
{"_id":3, "sku":"ijk", description:"product 3", "instock":60}
{"_id":4, "sku":"jkl", description:"product 4", "instock":70 }
{"_id":5, "sku":null, description:"Incomplete" }
{"_id":6}
```

A document with **sku** field equals null

A document with no **sku** field

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#pipe._S_lookup

$lookup stage example (cont'd)

```
{"_id":1, "item":"abc", "price":12,"quantity":2 }
{"_id":2, "item":"nosku", "price":20,"quantity":1 }
{"_id":3 }
```

```
{"_id":1, "sku":"abc", description:"product 1", "instock":120}
{"_id":2, "sku":"def", description:"product 2", "instock":80 }
{"_id":3, "sku":"ijk", description:"product 3", "instock":60}
{"_id":4, "sku":"jkl",                                   ck":70 }
{"_id":5, "sku":null,
{"_id":6}
```

Non es null and non exists field

```
{"_id":1, "item":"abc", "price":12,"quantity":2,
 "inventory_docs": [
    { "_id":1, "sku":"abc", description:"product 1", "instock":120 }] }
{"_id":2, "item":"nosku", "price":20,"quantity":1,
  "inventory_docs" : [] }
{"_id":3, "inventory_docs" : [
    { "_id" : 5, "sku" : null, "description" : "Incomplete" },
    { "_id" : 6 }]}
```

An empty array for no matching from other collection

Dealing with data of array type

- To aggregate (e.g. grouping) values in an array field, it is possible to flatten the array to access individual value
- **`$unwind`** stage flattens an array field from the input documents to output a document for *each* element. Each output document is the input document with the value of the array field replaced by the element.
  - **`{ $unwind: <field path> }`**
- Behaviour
  - Input document:

    { "_id" : 1, "item" : "ABC1", sizes: ["S", "
  - After `$unwind:`**`"$sizes"`**
  - Becomes 3 output documents:

    { "_id" : 1, "item" : "ABC1", "sizes" : "S" }

    { "_id" : 1, "item" : "ABC1", "sizes" : "M" }

    { "_id" : 1, "item" : "ABC1", "sizes" : "L" }

$unwind example

- Find the number of items that are available in each size

```
db.inventory.aggregate( [
{ $unwind : "$sizes" },
{ $group:{_id: “$sizes”, item_count: {$sum:1}} }
] )
```

$unwind then $group

```
{ "_id" : 1, "item" : "ABC", "sizes": [ "S", "M", "L"] }
{ "_id" : 2, "item" : "EFG", "sizes" : [ ] }
{ "_id" : 3, "item" : "IJK", "sizes": "M" }
{ "_id" : 4, "item" : "LMN" }
{ "_id" : 5, "item" : "XYZ", "sizes" : null }
```

```
{ $unwind : "$sizes" },
```

```
{ "_id" : 1
{ "_id" : 1, "item" : “ABC", "sizes": "M"}
{ "_id" : 1, "item" : “ABC", "
{ "_id" : 3, "item" : "IJK", "
```

```
{ $group:{_id: “$sizes”,
  item_count: {$sum:1}}
```

```
{ "_id" : "S", "item_count": 1}
{ "_id" : "M", "item_count": 2}
{ "_id" : "L", "item_count": 1}
```

Aggregation Operators

- A few aggregation stages allow us to add new fields or to given existing fields new values based on expression
  - In **`$group`** stage we can use various *operators* or *accumulators* to compute values for new fields
  - In **`$project`** stage we can use operators to compute values for new or exiting fields
- There are many r various data types to carry out common operation ta type
  - Arithmetic operators: **`$mod, $subtract, …`**
  - String operators: **`$concat, $split, $indexofBytes, …`**
  - Comparison operators: **`$gt, $gte, $lt, $lte,…`**
  - Set operators: **`$setEquals, $setIntersection, …`**
  - Boolean operators: **`$and, $or, $not, …`**
  - Array operators: **`$in, $size, ..`**

Aggregation vs. Query operators

- There is another set of operators that can be used in **`find/update/delete`** queries or the **`$match`** stage of an aggregation
  - E.g. **`$gt, $lt, $in, $all….`**
- The set is smaller and are different to the operators used in **`$group`** or **`$pro`**
- Some operators different syntax and slightly different interpretation d in aggregation.
  - E.g. **`$gt`** in query looks like
    `{age: {$gt:18}}`
  - **`$gt`** in **`$project`** stage looks like:
    `{over18: {$gt:[“$age”, 18]}}`

Aggregation Behaviour

- It operates on a single collection (before 3.2)
  - Join can be performed using a particular operator **$lookup**
- It logically passes the *entire* collection into the pipeline
- Early filtering can improve the performance
- **$match** and **$sort** e index if placed at the beginning of

Summary

- MongoDB stores data as BSON document
- Retrieving data from MongoDB are usually achieved through
  - **`find`** query
  - **`aggregate`** pipeline
- **`find`** query targ supports condition on any field
- **`aggregate`** pipeline is able to r collection(s)
- Both provides rich set of operators
- **`update/insert/delete`** operation guarantees document level atomicity
- None standard query API, set of operators are growing

References

- BSON types
  - https://docs.mongodb.com/manual/reference/bson-types/
- Aggregation Pipelines
  - https://docs.mongodb.com/manual/core/aggregation-pipeline/
- Aggregation ope
  - https://docs.mon perator/aggregation/