COMP5338 – Advanced Data Models

Week 3: MongoDB – Aggregation Framework

Dr. Ying Zhou
School of Computer Science

Outline

- Null type
- Aggregation
  - Single collection aggregation
  - Aggregation pipeline with multiple collection

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 here
- The interpretation of null is different depending on where it appears
- It might represents
  - The field exists, but has no value
  - The field does not exits
  - Or both
- This is different to giving a field an empty string “” as value

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

Null query example

- Collection revisions document sample

{ "_id" : ObjectId("5799843ee2cbe65d76ed919b"),
"title" : "Hillary_Clinton",
"timestamp" : "2016-07-23T02
"revid" : 731113635,
"user" : "BD2412",
"parentid" : 731113573,
"size" : 251742,
"minor" : ""}

```
{
    "_id" : ObjectId("5d42869d0c84336545f9b2d3"),
    "title" : "Hillary_Clinton",
    "timestamp" : ISODate("2016-07-01T19:33:39.000Z"),
    "parsedcomment" : "The religious affiliation of Hillary
    "revid" : 727871391,
    "user" : "Theologicalmess",
    "parentid" : 727749878,
    "size" : 246162
}
```

- 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 or major revision.

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

Querying for null or field existence

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

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.insertOne({title:"nulltest",
    "timestamp" : "2018-08-14T02:02:06Z",
    "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:false}})` can differentiate the two

Outline

- Null type
- Aggregation
  - Single collection aggregation pipeline
  - Aggregation pipeline with multiple collection

Aggregation

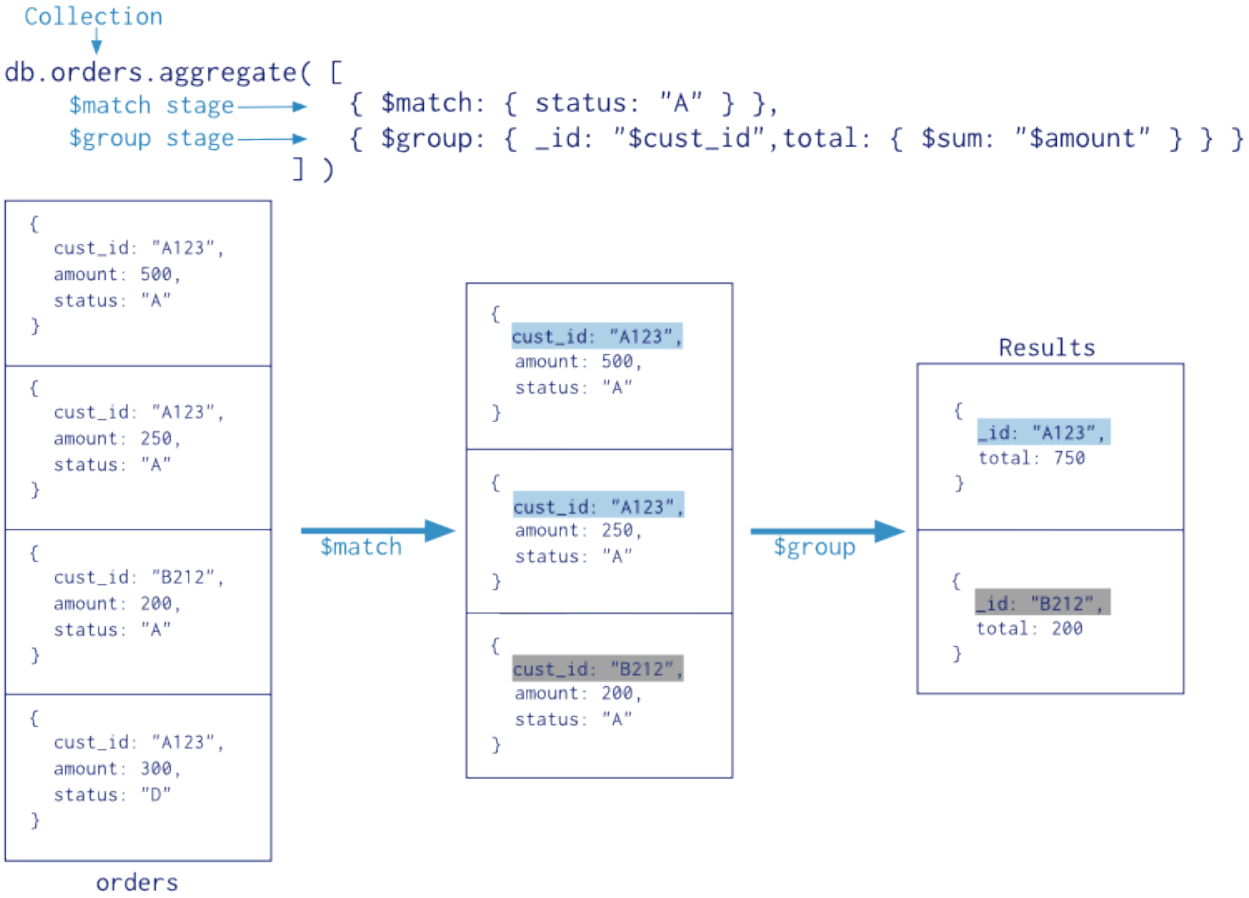
- Simple and relatively standard data analytics can be achieved through **aggregation**
  - Grouping, summing up value, counting, sorting, etc
  - Running on the DB engine instead of application layer
- Several options
  - Aggregation Pipeline
  - MapReduce
    - Through JavaScript Functions
    - Is able to do customized aggregations

Aggregation Pipeline

- Aggregation pipeline consists of multiple stages
  - Each stage transforms the incoming documents as expressed in the stage object
  - The stage object is enclosed in a pair of curly braces
  - The pipeline is an array of many stage objects.

```
db.collection.aggregate( [
        { stage name: {expression,…, expression} },
        { stage name: {expression,…, expression} },
        ...
        ] )
```

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`
- `$project`
- `$unwind`
- `$sort`
- `$skip`
- `$limit`
- `$count`
- `$sample`
- `$out`
- `$lookup`

$match stage

- **`$match:`** filters 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.aggregate([{$match:{size :{$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**
- Format:

```
{ $group:
    {_id:<expression>, // Group By Expression
     <field1>:{accumulator: <expression>},
            …}
}
```

  - The **`_id`** field of the output document has the value of the group key for each group
  - The other fields usually represent the statistics you want to produce for each group
  - One statistics per field
    - Total amount, average price, group size

$group stage (cont'd)

- **<Expression>** in **{_id:<expression>,**
  - **null** value, to specify the whole collection as a group
  - *field path* to to specify one or many fields as grouping key
    - Field name prefixed with $ sign in a pair of quotes
    - "**$title**", or "**$address.street**"
- **{accumulator: <expression>}**
  - There are predefined accumulators: `$sum`, `$avg`, `$first`, `$last`, etc
  - User defined accumulators can be used as well
  - Field path will be used in the <expression> if the accumulator returns value based on field values in the incoming document

$group stage example

- Find the earliest revision time in the whole collection

```
db.revisions.find({},{timestamp:1, _id:0})
        .sort({timestamp:1})
        .limit(1)
db.revisions.aggregate([
  {$group: {_id:null, earliest: {$min: "$timestamp"}}}
])
```

Sort in ascending order of timestamp

Accumulator

Field path as expression

Returns a single document

- Find the earliest revision time of each page in the collection

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

field path

Returns a document for each distinct title

$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

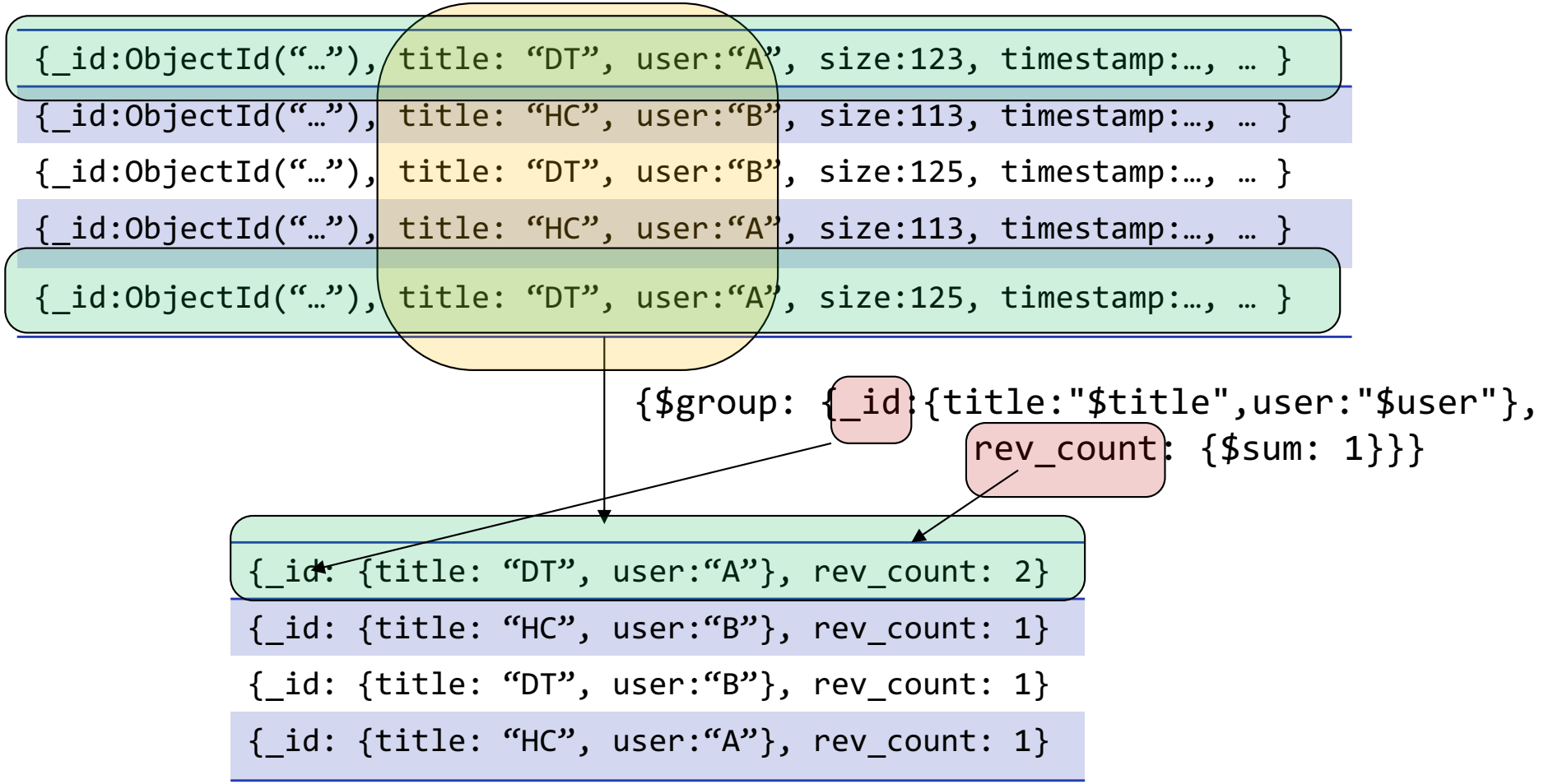
Composite type as _id

```
db.revisions.aggregate([
  {$group: {_id:{title:"$title",user:"$user"},
            rev_count: {$sum: 1}}}
])
```

accumulator number literal

Same effect as count

$group by more than one field



$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
- Example of two commands:

```
db.revisions.aggregate([
 {$group: {_id:"$title",
           revs: {$push:{user:"$user",timestamp:"$timestamp"}}}
}])

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

They have the same group key: $title

They have another field in addition to the group key
The other field is created with different accumulators

$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([
 {$group:
   {_id:"$title",
   revs:{$push:{user:"$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:…, … }
```

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

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

$project stage

- **`$project`**
  - **Restructure** 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: {<specification(s)}}
```

  - The specification can be an existing field name followed by a single value indicating the inclusion (1) or exclusion (0) of fields
  - Or it can be a field name (existing or new) followed 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

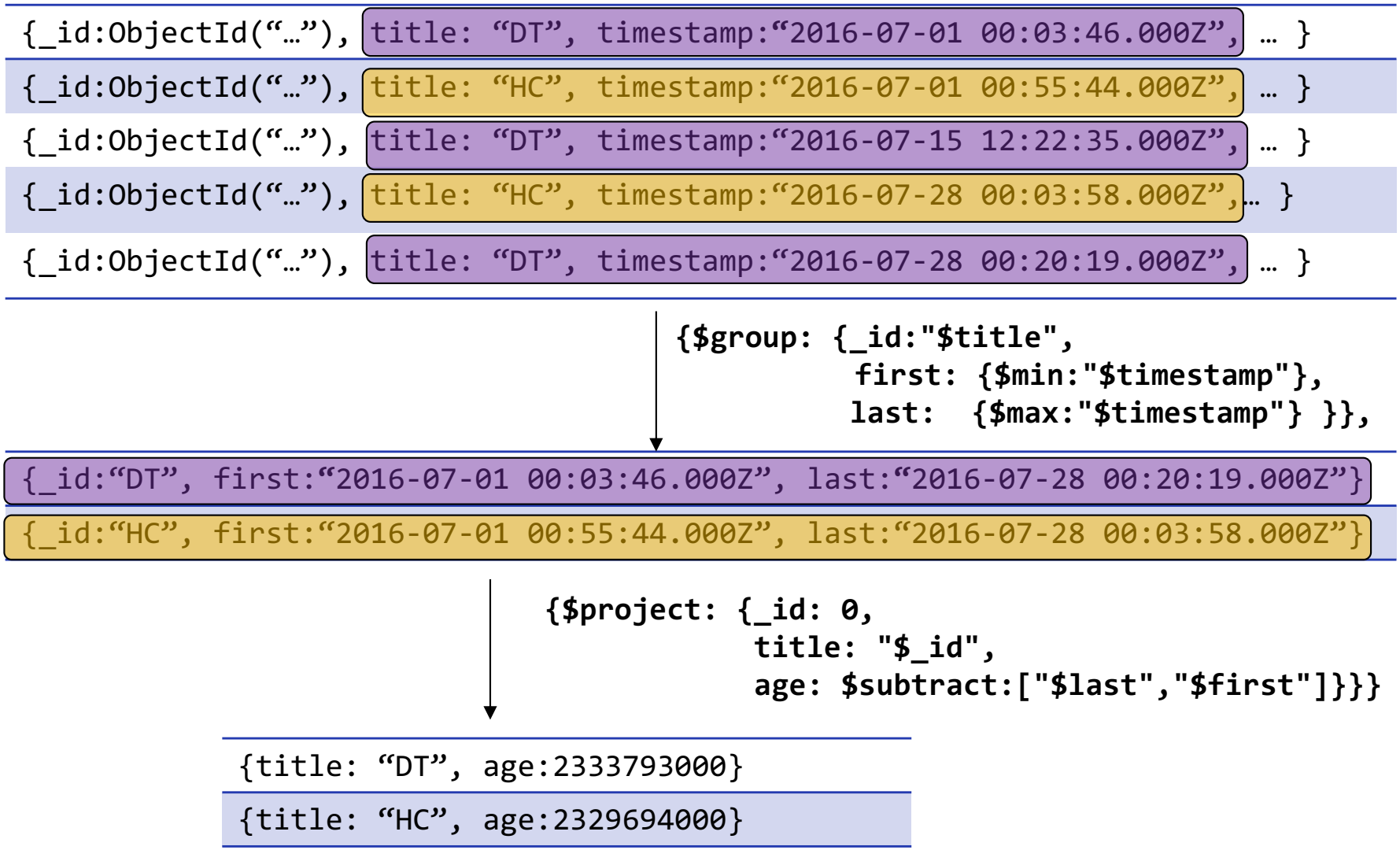
- Find the **age** of each title in the collection, where the **age** is defined as the duration between the last and the first revision of that title, assuming the timestamp is of ISODate type

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

Arithmetic expression operator, part of a large group of **Aggregation pipeline operator**

https://docs.mongodb.com/manual/reference/operator/aggregation/#arithmetic-expression-operators

$group then $project



We can combine multiple operators

```
db.revisions.aggregate([
{$group: {_id:"$title",
          first: {$min:"$timestamp"},
           last:  {$max:"$timestamp"} }},
{$project: {_id: 0,
            title: "$_id",
            age:{$divide:
                 [{$subtract:["$last","$first"]},
                 86400000]}}}
            age_unit: {$literal:"day"}}}
])
```

($last-$first)/86400000

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> }` or
  -

```
{
   $unwind:
    {
            path: <field path>,
            includeArrayIndex: <string>,
            preserveNullAndEmptyArrays: <boolean>
        }
   }
```

$unwind example

- Default behaviour
  - Input document:

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

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

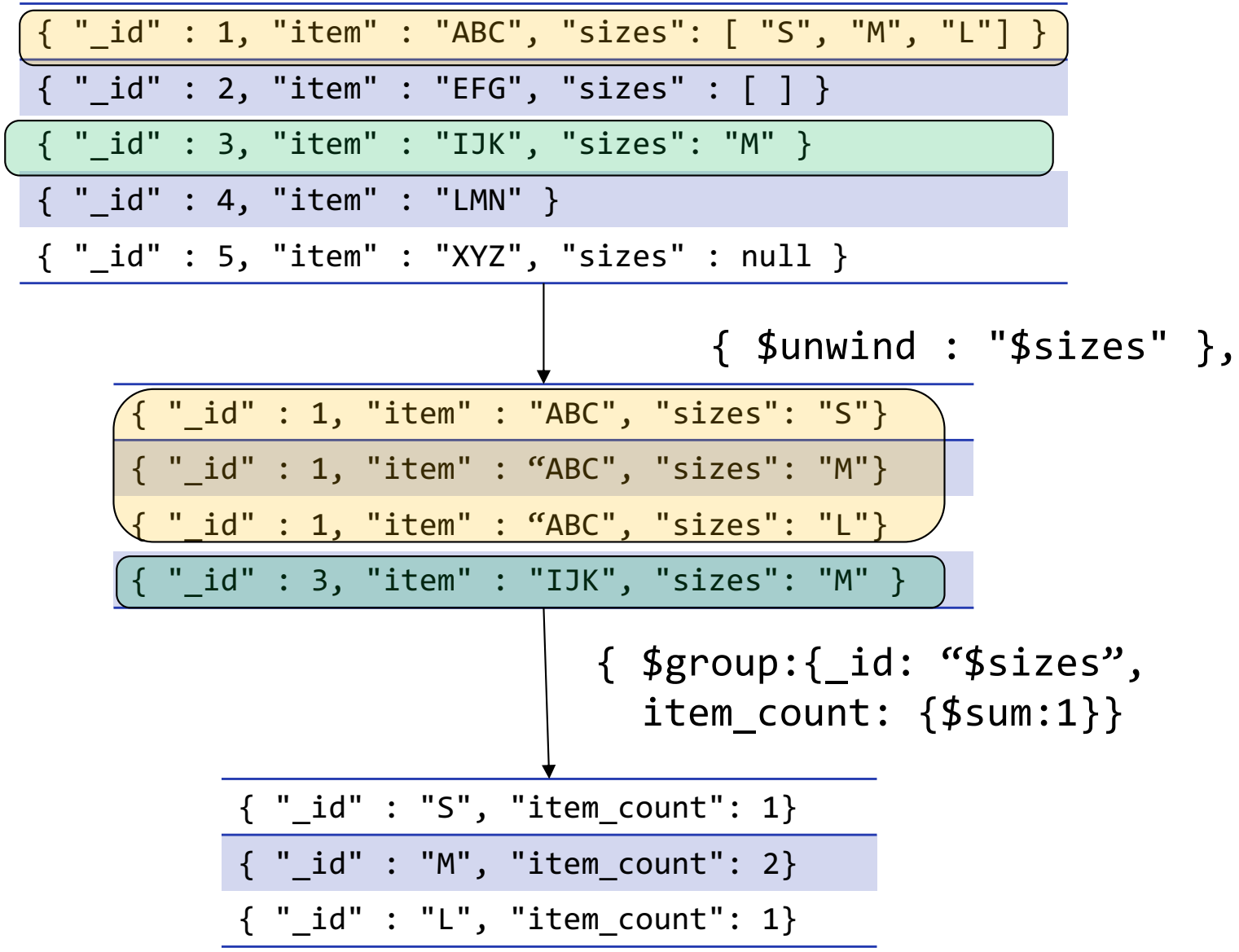
    { "_id" : 1, "item" : "ABC1", "sizes" : "M" }

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

- 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



`$sort`, `$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 format is similar to the skip modifier in **`find`** query
  - **`{ $skip: <positive integer> }`**
- **`$limit`** stage limits the number of documents passed to the next stage
  - The function and format is similar to the limit modifier 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
  - **{ $sample: { size: <positive integer> } }**
  - Different sampling approaches depending on the location of the stage and the size of the sample and the collection
  - May fail due to memory constraints
- The **$out** stage writes the documents in a given collection
  - should be the last one in the pipeline
  - **{ $out: "<output-collection>" }**

Aggregation Operators

- A few aggregation stages allow us to add new fields or to give 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 predefined operators for various data types to carry out common operations in that data type
  - Arithmetic operators: **`$mod, $log, $sqrt, $subtract, …`**
  - String operators: **`$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 **`$project`** stage
- Some operators look the same but have different syntax and slightly different interpretation in query and in aggregation.
  - E.g. **`$gt`** in find query looks like

    `{age: {$gt:18}}`
  - **`$gt`** in **`$project`** stage looks like:

    `{over18: {$gt:[“$age”, 18]}}`

Returns true or false

Outline

- Null type

- **Aggregation**
  - Single collection aggregation pipeline
  - **Aggregation pipeline with multiple collections**

`$lookup` stage

- **`$lookup`** stage is added since 3.2 to perform left outer join between two collections
  - The collection already in the pipeline (maybe after a few stages)
  - Another collection (could be the same one)
- For each *incoming document from the pipeline*, the `$lookup` stage adds a new **array field** whose elements are the matching documents *from the other collection*.
- A few different forms
  - Equality match
  - Join with other conditions
  - Join with uncorrelated sub-queries

`$lookup` stage (cont'd)

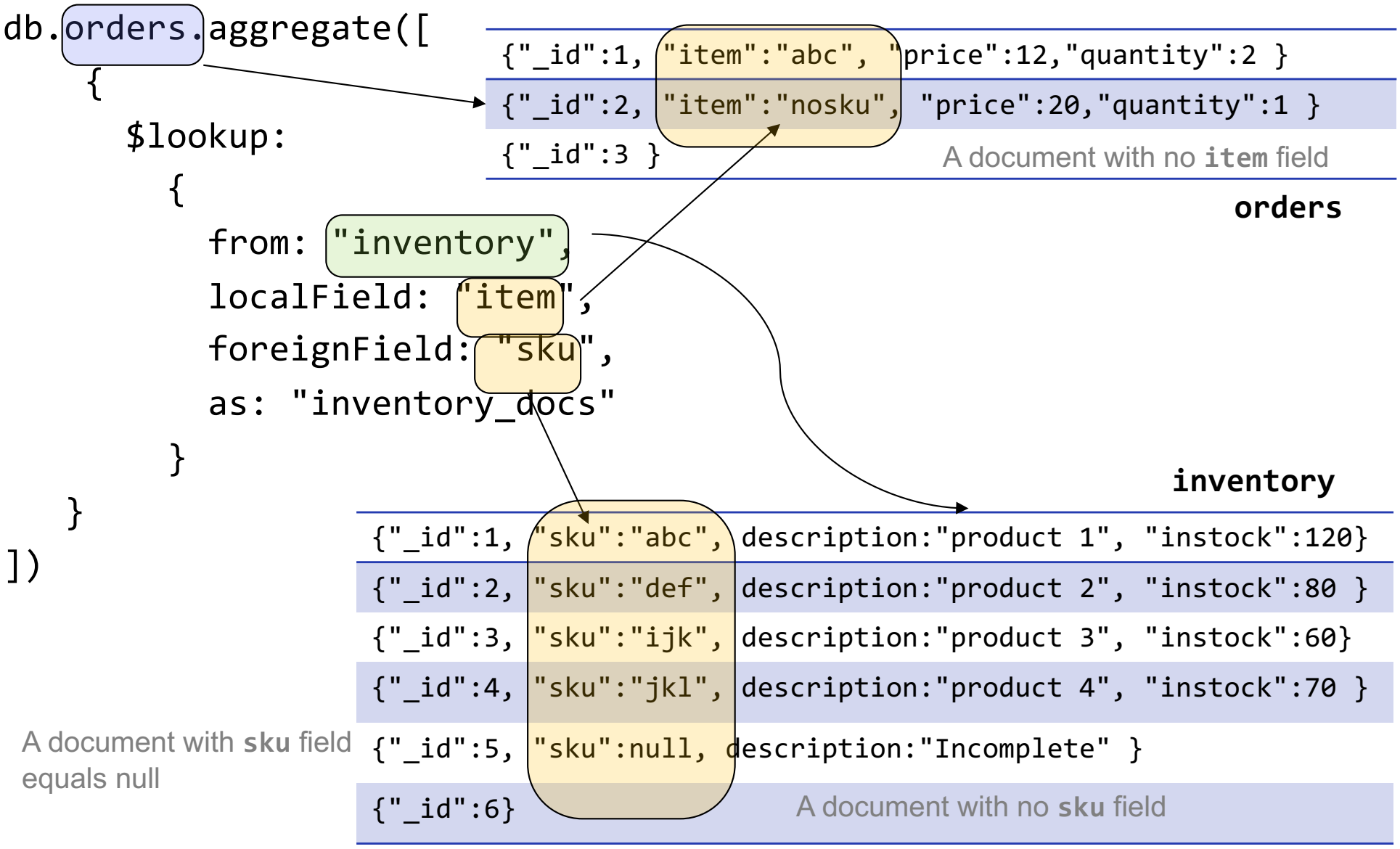
- The output of **`$lookup`** stage has the same number of documents as the previous stage
- Each document is augmented with an **array field** storing matching document(s) from the other collection
- The array could contain any number of documents depending on the match, including zero
- Missing local or foreign field is treated as having **`null`** value

Equality Match $lookup

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

```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (SELECT *
                               FROM <collection to join>
                               WHERE <foreignField>= <collection.localField>);
```

Equality match $lookup example



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

Equality match $lookup example (cont'd)

local

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

foreign

```
{"_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}
```

Non exists field matches null and non exists field

output

```
{"_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

Other format of $lookup

```
{
   $lookup:
     {
       from: <collection to join>,
       let: { <var_1>: <expression>, …, <var_n>: <expression> },
       pipeline: [ <pipeline to execute on the collection to join> ],
       as: <output array field>
     }
}
```

let: Optionally specifies variables to use in the pipeline field stages. Most likely the variable(s) may refer to field(s) in the local collection already in the pipeline

pipeline: Specifies the pipeline to run on the joined collection. The `pipeline` determines the resulting documents from the joined collection.

Multiple Joint Condition Example

orders collection

```
{ "_id" : 1, "item" : "almonds", "price" : 12, "ordered" : 2 }
{ "_id" : 2, "item" : "pecans", "price" : 20, "ordered" : 1 }
{ "_id" : 3, "item" : "cookies", "price" : 10, "ordered" : 60 }
```

warehouses collection

```
{ "_id" : 1, "stock_item" : "almonds", warehouse: "A", "instock" : 120 }
{ "_id" : 2, "stock_item" : "pecans", warehouse: "A", "instock" : 80 }
{ "_id" : 3, "stock_item" : "almonds", warehouse: "B", "instock" : 60 }
{ "_id" : 4, "stock_item" : "cookies", warehouse: "B", "instock" : 40 }
{ "_id" : 5, "stock_item" : "cookies", warehouse: "A", "instock" : 80 }
```

An ordered item may be stocked in multiple warehouses;

We want to find for each ordered item the warehouse with sufficient stock to cover the order

Multiple Joint Condition

- This query involves comparing two fields of the local and foreign documents:
  - “`item`” in **`orders`** should match ”`stock_item`” in **`warehouses`**
  - “`ordered`” in orders should be less than or equal to “`instock`” in warehouses

`orders` collection

```
{ "_id" : 1, "item" : "almonds", "price" : 12, "ordered" : 2 }
{ "_id" : 2, "item" : "pecans", "price" : 20, "ordered" : 1 }
{ "_id" : 3, "item" : "cookies", "price" : 10, "ordered" : 60 }
```

`warehouses` collection

```
{ "_id" : 1, "stock_item" : "almonds", warehouse: "A", "instock" : 120 }
{ "_id" : 2, "stock_item" : "pecans", warehouse: "A", "instock" : 80 }
{ "_id" : 3, "stock_item" : "almonds", warehouse: "B", "instock" : 60 }
{ "_id" : 4, "stock_item" : "cookies", warehouse: "B", "instock" : 40 }
{ "_id" : 5, "stock_item" : "cookies", warehouse: "A", "instock" : 80 }
```

Multiple Joint Condition $lookup

```
db.orders.aggregate([
   {
      $lookup:
         {
           from: "warehouses",
           let: { order_item: "$item", order_qty: "$ordered" },
           pipeline: [
              { $match:
                 { $expr:
                    { $and:
                       [
                         { $eq: [ "$stock_item",  "$$order_item" ] },
                         { $gte: [ "$instock", "$$order_qty" ] }
                       ]
                    }
                 }
              },
              { $project: { stock_item: 0, _id: 0 } }
           ],
           as: "stockdata"
         }
   }
])
```

This is the way to let the pipeline access local fields: use variable **order_item** to access the local document's **item** field; use variable **order_qty** to access the local document's **ordered** field

variables are accessed using "$$" prefix

This is the way to specify multiple condition

`$lookup` by default includes the entire matched foreign document in the array, we can use $project stage to get rid of some field

Matching document after the pipeline stage will be stored in this variable

Results

```
{ "_id" : 1,
  "item" : "almonds",
  "price" : 12,
  "ordered" : 2,
  "stockdata" : [
      { "warehouse" : "A", "instock" : 120 },
      { "warehouse" : "B", "instock" : 60 }
   ]
}
```

```
{ "_id" : 2,
  "item" : "pecans",
   "price" : 20,
   "ordered" : 1,
   "stockdata" : [
       { "warehouse" : "A", "instock" : 80 }
    ]
}
```

```
{ "_id" : 3,
  "item" : "cookies",
  "price" : 10,
  "ordered" : 60,
  "stockdata" : [
      { "warehouse" : "A", "instock" : 80 }
   ]
}
```

Uncorrelated Subquery Example

absences collection

```
{
 "_id" : 1,
 "student" : "Ann Aardvark",
  sickdays: [ "2018-05-01", 2018-08-23"]
}
```

```
{
  "_id" : 2,
  "student" : "Zoe Zebra",
  sickdays: [“2018-02-01", 2018-05-23") ]
}
```

holidays collection

```
{ "_id" : 1, year: 2018, name: "New Years", date: "2018-01-01" }
{ "_id" : 2, year: 2018, name: "Pi Day", date: 2018-03-14" }
{ "_id" : 3, year: 2018, name: "Ice Cream Day", date: "2018-07-15"}
{ "_id" : 4, year: 2017, name: "New Years", date: "2017-01-01" }
{ "_id" : 5, year: 2017, name: "Ice Cream Day", date: "2017-07-16”}
```

We want to include all 2018 public holidays in the **absences** collection

Uncorrelated Subquery $lookup

```
db.absences.aggregate([
   {
      $lookup:
         {
           from: "holidays",
           pipeline: [
              { $match: { year: 2018 } },
              { $project: { _id: 0,
                      date: { name: "$name", date: "$date" } } },
              { $replaceRoot: { newRoot: "$date" } }
           ],
           as: "holidays"
         }
    }
])
```

The inner pipeline selects documents from **holidays** collection based on a condition unrelated with the local collection

The **$project** and **$replaceRoot** change the structure of the inner pipeline output documents

Has the same effect as: { **$project: { _id: 0, year:0 }** } in this case

$replaceRoot and similar "project" like stage are useful for promoting an embedded document at root level

Results

```
{
 "_id" : 1,
 "student" : "Ann Aardvark",
  sickdays: [ "2018-05-01", 2018-08-23"],
  "holidays" : [
      { "name" : "New Years", "date" : " 2018-01-01") },
      { "name" : "Pi Day", "date" :  "2018-03-14") },
      { "name" : "Ice Cream Day", "date" : "2018-07-15"}
  ]
}
```

```
{
  "_id" : 2,
  "student" : "Zoe Zebra",
  sickdays: [“2018-02-01", 2018-05-23") ],
  "holidays" : [
      { "name" : "New Years", "date" : " 2018-01-01") },
      { "name" : "Pi Day", "date" :  "2018-03-14") },
      { "name" : "Ice Cream Day", "date" : "2018-07-15"}
  ]
}
```

References

- BSON types
  - https://docs.mongodb.com/manual/reference/bson-types/
- Querying for Null or Missing Field
  - https://docs.mongodb.com/manual/tutorial/query-for-null-fields/index.html
- Aggregation Pipelines
  - https://docs.mongodb.com/manual/core/aggregation-pipeline/
- Aggregation operators
  - https://docs.mongodb.com/manual/reference/operator/aggregation/