# mongoDB

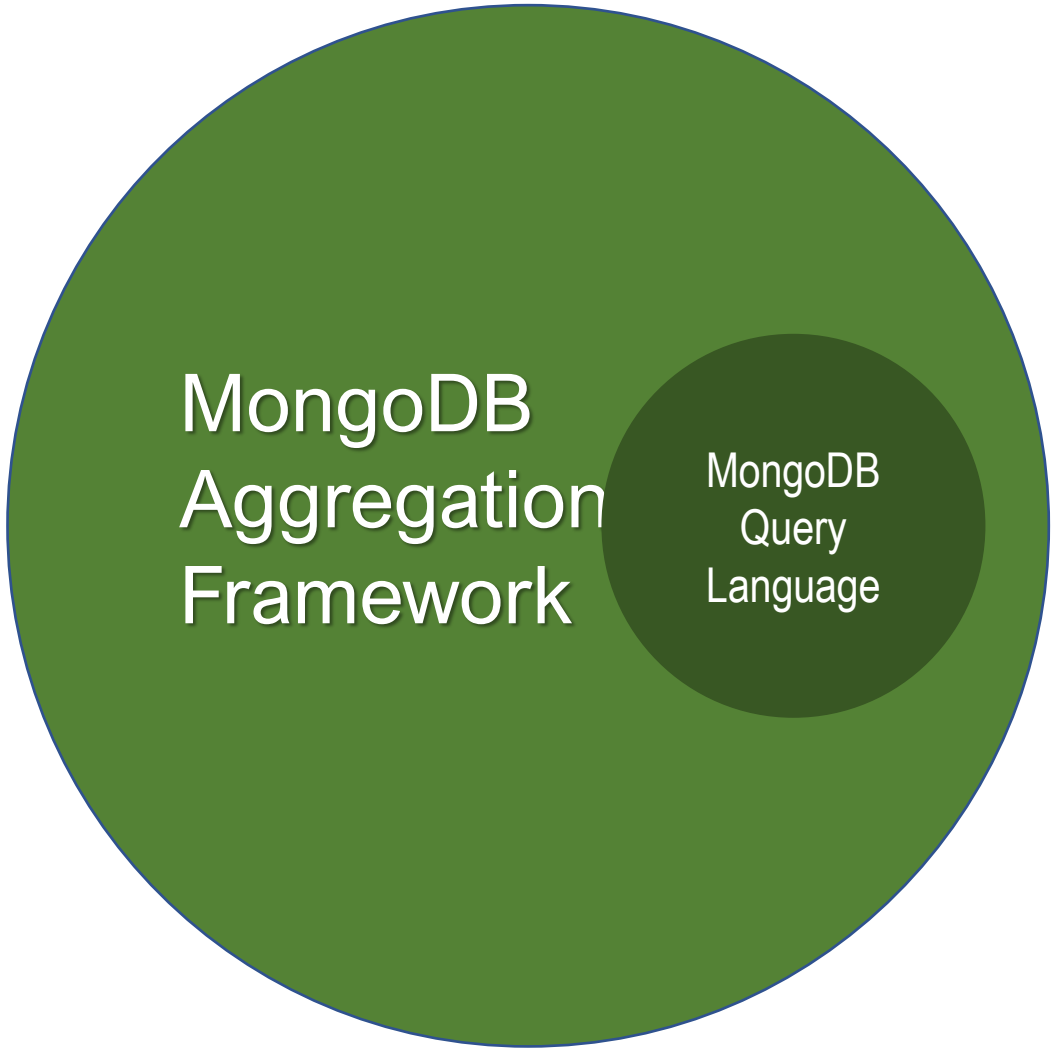
**Bộ môn: Kỹ Thuật Phần Mềm**

Giáo viên: **Trần Thế Trung.**
Email: tranthetrung@iuh.edu.vn

# MongoDB
# Aggregation Framework

1. Giới thiệu về Aggregation Framework.
2. Cấu trúc và cú pháp sử dụng Aggregation.
3. Tạo và sử dụng View.

# What is the MongoDB aggregation framework?




- In its simplest form, is just another way to query data in MongoDB
- Everything we know how to do using the MongoDB query language (MQL) can also be done using the Aggregation framework

# Why do we need Aggregation Framework

- We might want to aggregate, as in group or modify our data in some way, instead of always just filtering for the right documents.
- We can also calculate using aggregation.
- With MQL: we can filter and update data
- With Aggregation Framework: we can compute and reshape data

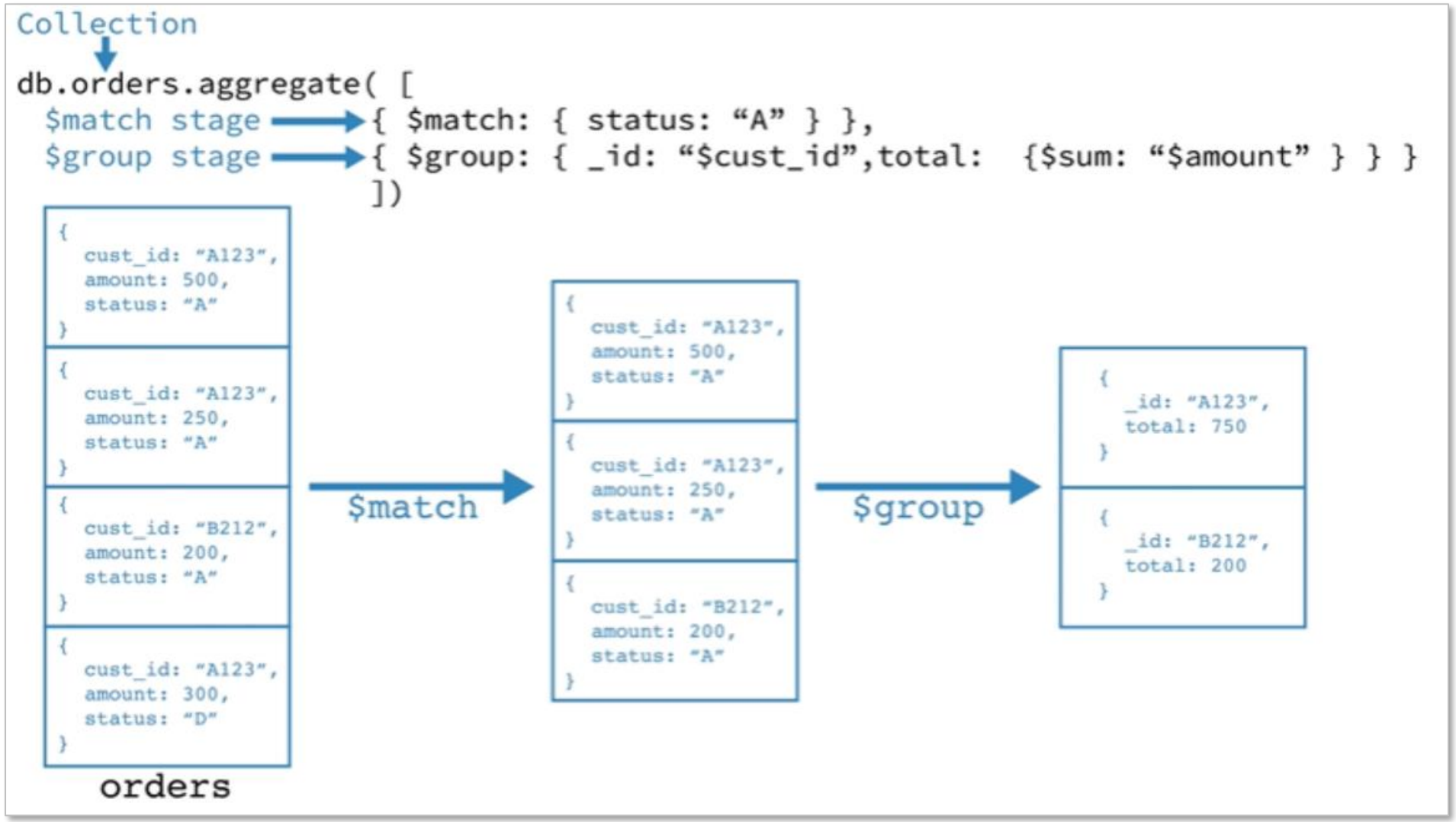
# Example

Let's find all documents that have Wi-Fi as one of the amenities, only include the price and address in the resulting cursor:

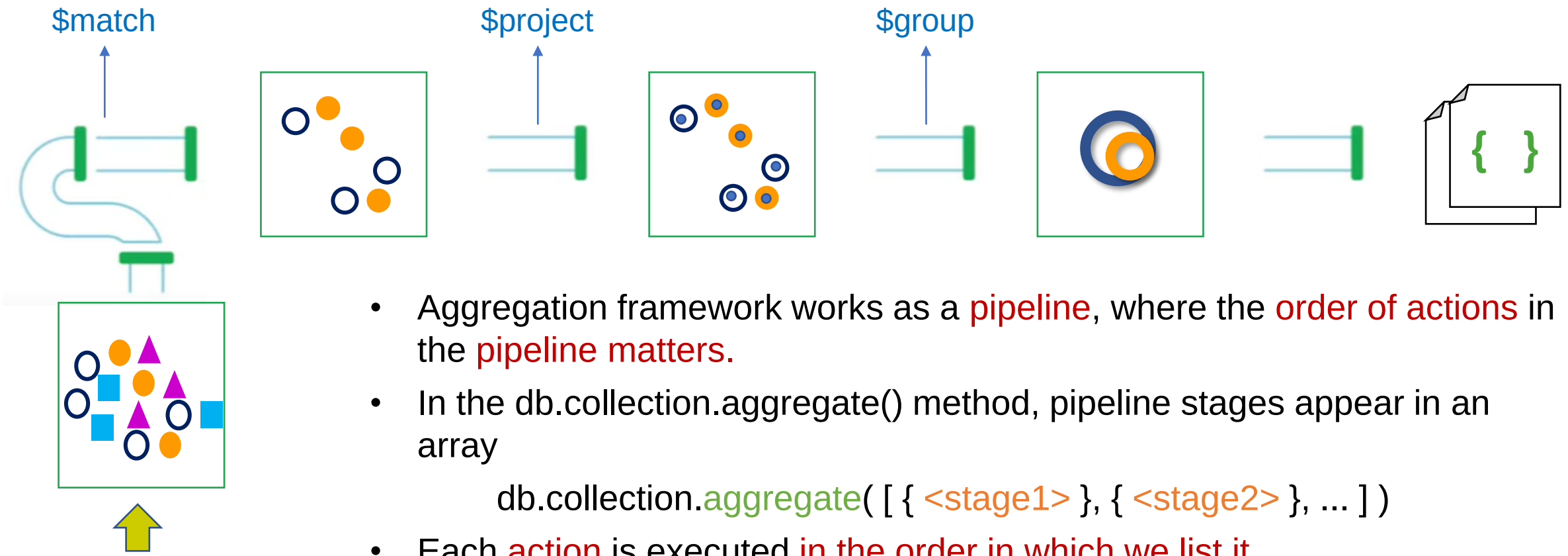
```
db.listingsAndReviews.find(
                        { amenities : 'Wifi' },
                        { price : 1, address : 1, _id : 0 }
                    )

db.listingsAndReviews.aggregate(
                        [ { $match : { amenities : 'Wifi' } },
                        { $project : { price : 1, address : 1, _id : 0 } } ]
                    )
```

# Pipeline


Collection
db.orders.aggregate( [
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: {$sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

# Pipeline




- Aggregation framework works as a pipeline, where the order of actions in the pipeline matters.
- In the db.collection.aggregate() method, pipeline stages appear in an array

  db.collection.aggregate( [ { <stage1> }, { <stage2> }, ... ] )
- Each action is executed in the order in which we list it.

# Aggregation Structure and Syntax

**Syntax**: db.collection.aggregate( [ { <stage1> }, { <stage2> }, ... ], {option})

- Each stage is a JSON object of key value pairs.
- Options may be passed in. For example, specifying whether to allow disk use for large aggregations, or to view the explain plan of the aggregation to see whether it is using indexes.

**Example**:

```
db.solarSystem.aggregate( [
        { $match : { atmosphericComposition: { $in : [/O2/] }, meanTemperature: { $gte : -40, $lte :40} } },
        { $project : { _id : 0, name : 1, hasMoons: { $gt : [ '$numberOfMoons', 0 ] } } }
    ],
    { allowDiskUse : true }
)
```

- Pipelines are always an array of one or more stages.
- Stages are composed of one or more aggregation operators or expressions
- Expressions may take a single argument or an array of arguments *(Read More)*

# Common Pipeline States/Expression

| Method | Description |
|---|---|
| $match() | Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage. *(Read more)* |
| $project() | Reshapes each document in the stream, such as by adding new fields or removing existing fields. *(Read more)* |
| $group() | Group documents in collection, which can be used for statistics. *(Read more)* |
| $unwind() | Deconstructs an array field from the input documents to output a document for each element. *(Read more)* |
| $lookup() | Performs a left outer join to an unsharded collection in the same database to filter in documents from the "joined" collection for processing. *(Read more)* |
| $redact | Restricts the contents of the documents based on information stored in the documents themselves. *(Read more)* |
| $out | Takes the documents returned by the aggregation pipeline and writes them to a specified collection. *(Read more)* |
| $merge | Writes the results of the aggregation pipeline to a specified collection. *(Read more)* |
| *self study: $sort(), $limit(), $skip(), … (Read More)* | |

# $match stage

**Syntax**: { $match : { <query> } }

- Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage.
- Place the $match as early in the aggregation pipeline as possible.
- $match can be used multiple times in pipeline.
- $match uses standard MongoDB query operators.
- you cannot use $where with $match.

# $match stage

- **Example**:

```
db. sinhvien.aggregate( [ {  $match : { ten: { $eq: 'Nở' } } } ] )

db. sinhvien.aggregate( [
        { $match : { ten: { $eq: 'Nở' } } },
        {  $count: 'TongSoSV' }
    ] )

db.sinhvien.aggregate( [
        { $match : {
            $and : [
                { 'lienLac.email' : 'teo@gmail.com' },
                { _id : { $eq : '57' } }
            ]
        }
    } ] )
```

# $project stage - Shaping documents

**Syntax**: { $project : { <specification(s)> } }

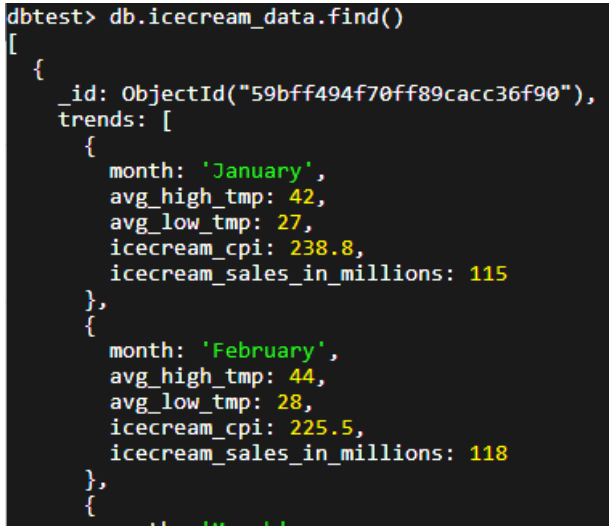
- With $project state we can selectively remove and retain fields and also reassign existing field values and derive entirely new fields.
- $project can be used as many times as required with an aggregation pipeline

**Example:**

```
db.solarSystem.aggregate( [ { $project : { _id : 0, name : 1, gravity: 1} } ] )
db.solarSystem.aggregate( [ { $project : { _id : 0, name : 1, 'gravity.value' : 1} } ] )
db.solarSystem.aggregate( [ { $project : { _id : 0, name : 1, surfacegravity: '$gravity.value' } } ] )
db.solarSystem.aggregate( [ { $project : { _id : 0, name : 1, new_surfacegravity: {
                                                    $multiply: [
                                                        { $divide: [ '$gravity.value', 10 ] }, 100
                                                    ]
                                                } } } ] )
```

# Accumulator Expression with $project stage

- Accumulator expressions within $project work over an array within the given document
- Some of accumulator expressions: $avg, $min, $max, $sum, …
- We're going to explore 'icecream_data' collection in dbtest database




**Example**:

```
db.icecream_data.aggregate( [ { $project : { max_high: { $max: '$trends.avg_high_tmp' } } } ] )
```

[ { _id : ObjectId('59bff494f70ff89cacc36f90'), max_high: 87 } ]

```
db.icecream_data.aggregate( [ { $project : { min_low: { $min: '$trends.avg_low_tmp' } } } ] )
```

[ { _id : ObjectId('59bff494f70ff89cacc36f90'), min_low: 27 } ]

```
db.icecream_data.aggregate( [ { $project : { average_sale: { $avg: '$trends.icecream_sales_in_millions' } } } ] )
```

[ { _id : ObjectId('59bff494f70ff89cacc36f90'), average_sale: 133.41666666666666 } ]

# $group stage

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

| Field | Description |
|---|---|
| _id | Required. If you specify an _id value of null, or any other constant value, the $group stage calculates accumulated values for all the input documents as a whole. |
| field | Optional. Computed using the accumulator operators. |

- Groups input documents by the specified _id expression and for each distinct grouping, outputs a document.
- The _id field of each output document contains the unique group by value.
- The output documents can also contain computed fields that hold the values of some accumulator expression.

# $group stage

**Example:**

```
db.movies.aggregate( [ { $group : { _id : '$year', 'numFilmsThisYear' : { $sum: 1 } } } ] )
```

access the value of year's field

```
[
  { _id: 1901, numFilmsThisYear: 2 },
  { _id: 1916, numFilmsThisYear: 29 },
  { _id: 1963, numFilmsThisYear: 199 },
  { _id: 1925, numFilmsThisYear: 37 },
  { _id: 1943, numFilmsThisYear: 140 },
  { _id: 1934, numFilmsThisYear: 138 },
```

```
db.movies.aggregate( [
                    { $group : {
                                _id : '$year' ,
                                'numFilmsThisYear': { $sum: 1 }
                        } },
                    { $sort: { _id : 1} }
                ] )
```

```
[
  { _id: 1892, numFilmsThisYear: 1 },
  { _id: 1893, numFilmsThisYear: 1 },
  { _id: 1894, numFilmsThisYear: 1 },
  { _id: 1895, numFilmsThisYear: 2 },
  { _id: 1896, numFilmsThisYear: 5 },
```

# $group stage

- Grouping as before, then sorting in descending order based on the count

```
db.movies.aggregate( [
                { $group : { _id : '$year', count : { $sum : 1 } } },
                { $sort : { count : -1} }
            ] )
```

```
[
  { _id: 1972, count: 338 },
  { _id: 1971, count: 333 },
  { _id: 1970, count: 311 },
  { _id: 1973, count: 303 },
  { _id: 1974, count: 301 },
  { _id: 1976, count: 284 },
```

- Grouping on the number of directors a film has, demonstrating that we have to validate types to protect some expressions

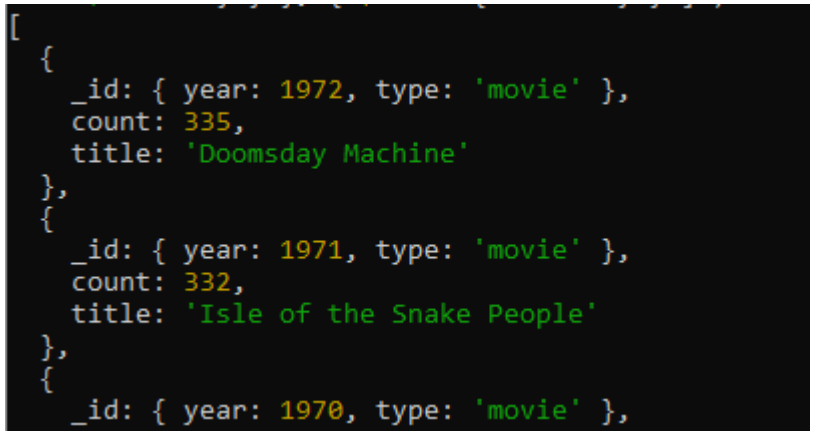
```
db.movies.aggregate( [
                { $group : {
                      _id : { numDirectors : { $cond : [ { $isArray : '$directors' }, { $size : '$directors' }, 0 ] } },
                      numFilms : { $sum : 1 },  averageMetacritic: { $avg : '$metacritic' } } },
                { $sort : { '_id.numDirectors' : -1 } }
            ] )
```

```
[
  { _id: { numDirectors: 11 }, numFilms: 2, averageMetacritic: null },
  { _id: { numDirectors: 10 }, numFilms: 1, averageMetacritic: null },
  { _id: { numDirectors: 9 }, numFilms: 1, averageMetacritic: null },
  { _id: { numDirectors: 8 }, numFilms: 3, averageMetacritic: null },
  { _id: { numDirectors: 7 }, numFilms: 9, averageMetacritic: null },
  { _id: { numDirectors: 6 }, numFilms: 5, averageMetacritic: null },
```

# $group stage

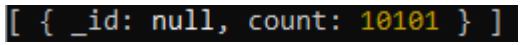
- Grouping on multiple columns

```
db.movies.aggregate( [
                { $group :
                  { _id : { year : '$year', type : '$type' },
                    count : { $sum : 1 }, title : { $first : '$title' }
                } },
                { $sort : { count : -1 } }
            ] )
```



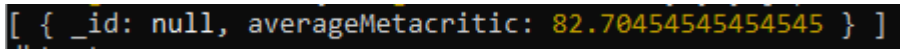

- Showing how to group all documents together. By convention, we use null or an empty string

```
db.movies.aggregate( [ { $group : { _id : null,  count : { $sum: 1 } } } ] )
```




- Filtering results to only get documents with a numeric metacritic value

```
db.movies.aggregate( [
                { $match : { metacritic : { $gte : 0 } } },
                { $group : { _id : null, averageMetacritic: { $avg: '$metacritic' } } }
            ] )
```



# $unwind stage

**Syntax**: { $unwind : <field path> }

- Deconstructs 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.

**Example**:

{ Title : *'The Martian'* , genres : [ *'Action'*, *'Adventure'*, *'Sci-Fi'* ] }
{ Title : *'Batman Begins'* , genres : [ *'Action'* , *'Adventure'* ] }

$unwind : '$genres'

{ Title : *'The Martian'* , genres : *'Action'* }
{ Title : *'The Martian'* , genres : *'Adventure'* }
{ Title : *'The Martian'* , genres : *'Sci_Fi'* }
{ Title : *'Batman Begins'* , genres : *'Action'* }
{ Title : *'Batman Begins'* , genres : *'Adventure'* }

# $unwind stage

- How to to group on year and genres of Movies collection?

```
db.movies.aggregate( [
                { $unwind : '$genres' },
                { $group : { _id : {year : '$year', genre : '$genres' }, numFilms : { $sum: 1 }, } },
                { $sort : { '_id.genre' : 1, numFilms: -1} }
        ] )
```

- Finding the top rated genres per year from 1990 to 2015...

```
db.movies.aggregate( [
                { $match : { 'imdb.rating' : { $gt : 0 }, year: { $gte : 1990, $lte : 2015 }, runtime : { $gte : 90 } } },
                { $unwind : '$genres' },
                { $group : { _id : { year : '$year' , genre : '$genres' }, average_rating : { $avg: '$imdb.rating' } } },
                { $sort : { '_id.year' : -1, average_rating : -1 } }
        ] )
```

# $unwind stage

**Recap on a few things:**

- $unwind only works on an array of values.
- Using unwind on large collections with big documents may lead to performance issues.

# $lookup stage

**Syntax**: equality match with a single join condition

```
{  $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>
);
```

- Performs a left outer join to an unsharded collection in the same database to filter in documents from the 'joined' collection for processing.
- To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the 'joined' collection.

# $lookup stage

**Example:**

```
aggregation> db.air_airlines.find({name: 'Air Berlin'})
[
  {
    _id: ObjectId("56e9b497732b6122f8790355"),
    airline: 214,
    name: 'Air Berlin',
    alias: 'AB',
    iata: 'BER',
    icao: 'AIR BERLIN',
    active: 'Y',
    country: 'Germany',
    base: 'KTE'
  }
]
```

```
aggregation> db.air_alliances.findOne()
{
  _id: ObjectId("5980bef9a39d0ba3c650ae9d"),
  name: 'OneWorld',
  airlines: [
    'Air Berlin',          'American Airlines',
    'British Airways',     'Cathay Pacific',
    'Finnair',             'Iberia Airlines',
    'Japan Airlines',      'LATAM Chile',
    'LATAM Brasil',        'Malasya Airlines',
    'Canadian Airlines',   'Quantas',
    'Qatar Airways',       'Royal Jordainian',
    'SriLanka Airlines',   'S7 Airlines'
  ]
}
```

Collection: **air_airlines**

Collection: **air_alliances**

```
db.air_alliances.aggregate( [
                    { $lookup : {
                          from : 'air_airlines',
                          localField : 'airlines',
                          foreignField : 'name',
                          as : 'airlines' }
                    }
                ] )
```

```
[
  {
    _id: ObjectId("5980bef9a39d0ba3c650ae9d"),
    name: 'OneWorld',
    airlines: [
      {
        _id: ObjectId("56e9b497732b6122f87907c8"),
        airline: 1355,
        name: 'British Airways',
        alias: 'BA',
        iata: 'BAW',
        icao: 'SPEEDBIRD',
        active: 'Y',
        country: 'United Kingdom',
        base: 'VDA'
      },
      {
        _id: ObjectId("56e9b497732b6122f87908cd"),
        airline: 1615,
        name: 'Canadian Airlines',
        alias: 'CP',
        iata: 'CDN',
        icao: 'CANADIAN',
        active: 'Y',
        country: 'Canada',
        base: 'LVI'
      },
      {
```

# $lookup stage

**Example:**

```
db.air_alliances.aggregate( [
                    { $match : { name : 'SkyTeam' } },
                    { $lookup : {
                          from : 'air_airlines',
                          localField : 'airlines',
                          foreignField : 'name',
                          as : 'airlines' }
                    }
                ] )
```


```
{
  _id: ObjectId("5980bef9a39d0ba3c650ae9c"),
  name: 'SkyTeam',
  airlines: [
    {
      _id: ObjectId("56e9b497732b6122f87902d9"),
      airline: 90,
      name: 'Air Europa',
      alias: 'UX',
      iata: 'AEA',
      icao: 'EUROPA',
      active: 'Y',
      country: 'Spain',
      base: 'RPR'
    },
    {
      _id: ObjectId("56e9b497732b6122f879095a"),
      airline: 1756,
      name: 'China Airlines',
      alias: 'CI',
      iata: 'CAL',
      icao: 'DYNASTY',
      active: 'Y',
      country: 'Taiwan',
      base: 'AGN'
    },
    {
      _id: ObjectId("56e9b497732b6122f879095c"),
      airline: 1758,
```

# $lookup stage

**Syntax**: correlated subqueries using concise syntax (*New in version 5.0*)

```
{  $lookup : {
        from : <collection to join>,
        localField : <field from local collection's documents>,
        foreignField : <field from foreign collection's documents>,
        let : { <var_1>: <expression>, …, <var_n>: <expression> },
        pipeline : [ <pipeline to run> ],
        as : <output array field>
    }
}
```

```
SELECT *, <output array field>
FROM localCollection
WHERE <output array field> IN (
   SELECT <documents as determined from the pipeline>
   FROM <foreignCollection>
   WHERE <foreignCollection.foreignField> = <localCollection.localField>
   AND <pipeline match condition>
);
```

- **let**: Optional. Specifies the variables to use in the pipeline stages. Use the variable expressions to access the document fields that are input to the pipeline
- **pipeline**: determines the resulting documents from the joined collection. To return all documents, specify an empty pipeline []. The pipeline cannot directly access the document fields. Instead, define variables for the document fields using the let option and then reference the variables in the pipeline stages

# $lookup stage

**Examples:**

```
db.air_alliances.aggregate( [
                    { $lookup : {
                                from : 'air_airlines',
                                localField : 'airlines',
                                foreignField : 'name',
                                pipeline: [ { $project : { _id : 0, name : 1, country : 1 } } ],
                                as : 'airlines' } }
                ] )
```

# $lookup stage

**Examples:**

```
aggregation> db.Orders.find()
[
  { _id: 1, item: 'filet', restaurant_name: 'American Steak House' },
  {
    _id: 2,
    item: 'cheese pizza',
    restaurant_name: 'Honest John Pizza',
    drink: 'lemonade'
  },
  {
    _id: 3,
    item: 'cheese pizza',
    restaurant_name: 'Honest John Pizza',
    drink: 'soda'
  }
]
```

Collection: **Orders**

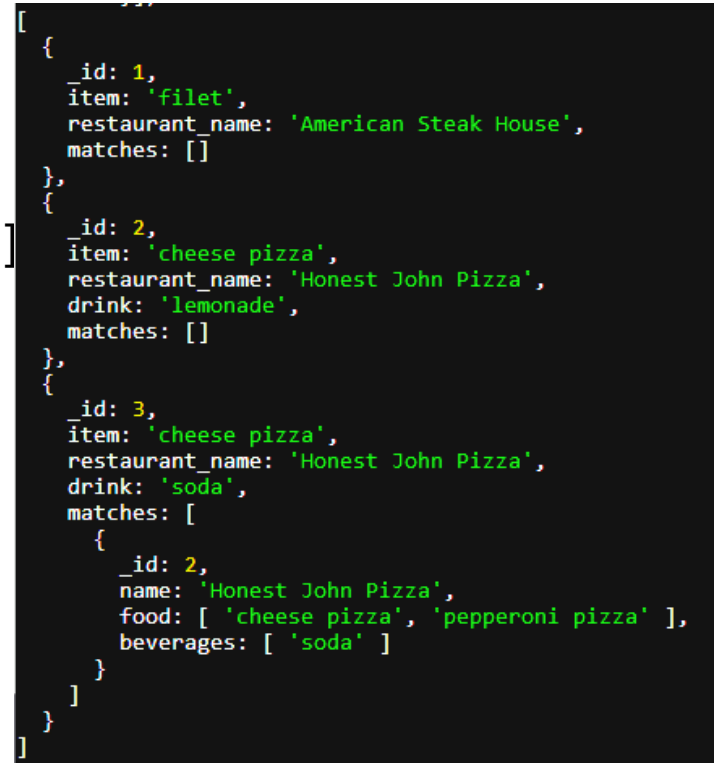
```
aggregation> db.Restaurants.find()
[
  {
    _id: 1,
    name: 'American Steak House',
    food: [ 'filet', 'sirloin' ],
    beverages: [ 'beer', 'wine' ]
  },
  {
    _id: 2,
    name: 'Honest John Pizza',
    food: [ 'cheese pizza', 'pepperoni pizza' ],
    beverages: [ 'soda' ]
  }
]
```

Collection: **Restaurants**

# $lookup stage

**Examples:**

```
db.Orders.aggregate( [
                { $lookup : {
                        from : 'Restaurants',
                        localField : 'restaurant_name',
                        foreignField : 'name',
                        let : { orders_drink : '$drink' },
                        pipeline : [ {
                                $match : { $expr : { $in : [ '$$orders_drink', '$beverages' ] }
                            } ],
                        as : 'matches'
                }
        } ] )
```



```
[
  {
    _id: 1,
    item: 'filet',
    restaurant_name: 'American Steak House',
    matches: []
  },
  {
    _id: 2,
    item: 'cheese pizza',
    restaurant_name: 'Honest John Pizza',
    drink: 'lemonade',
    matches: []
  },
  {
    _id: 3,
    item: 'cheese pizza',
    restaurant_name: 'Honest John Pizza',
    drink: 'soda',
    matches: [
      {
        _id: 2,
        name: 'Honest John Pizza',
        food: [ 'cheese pizza', 'pepperoni pizza' ],
        beverages: [ 'soda' ]
      }
    ]
  }
]
```

*(Read more about $expr)*

# $lookup stage

**Syntax**: perform multiple joins and a correlated subquery with $lookup

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

**Example**: list of warehouses with product quantity greater than or equal to the ordered product quantity.

```
aggregation> db.item_orders.find()
[
  { _id: 1, item: 'almonds', price: 12, ordered: 2 },
  { _id: 2, item: 'pecans', price: 20, ordered: 1 },
  { _id: 3, item: 'cookies', price: 10, ordered: 60 }
]
```

Collection: **item_orders**

```
aggregation> db.warehouses.find()
[
  { _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 }
]
```

Collection: **warehouses**

# $lookup stage

**Example**: list of warehouses with product quantity greater than or equal to the ordered product quantity.

```
db.item_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' }
                    } ] )
```

```
[
  {
    _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 } ]
  }
]
```

# $redact stage

**Syntax**: { $redact : <expression> } *(Read more about <expression>)*

Restricts the contents of the documents based on information stored in the documents themselves.

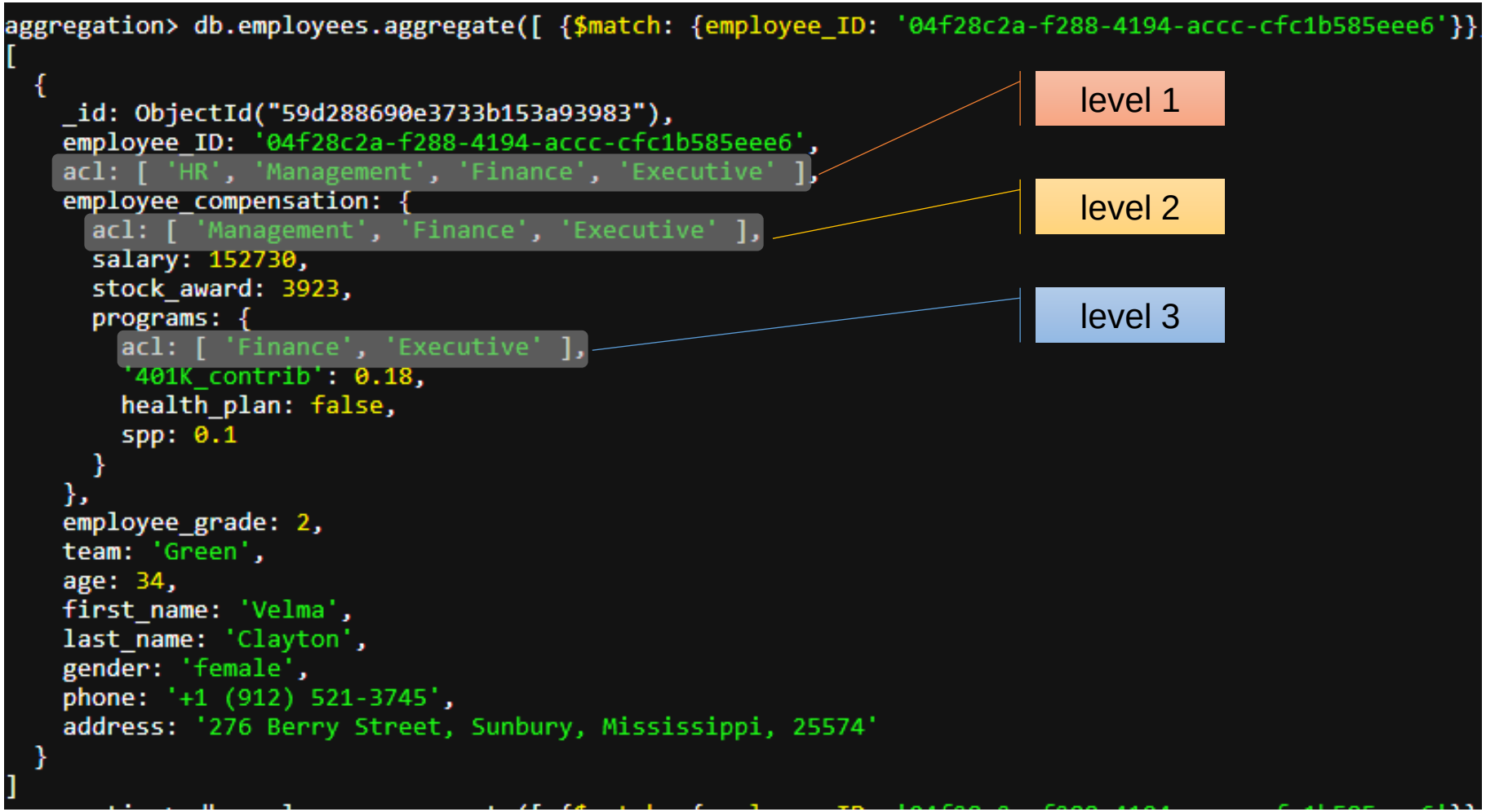
The argument can be any valid expression as long as it resolves to the $$descend, $$prune, or $$keep system variables.

| System Variable | Description |
|---|---|
| $$descend | $redact returns the fields at the current document level, excluding embedded documents |
| $$prune | $redact excludes all fields at this current document/embedded document level, without further inspection of any of the excluded fields |
| $$keep | $redact returns or keeps all fields at this current document/embedded document level, without further inspection of the fields at this level |

# $redact stage

**Examples:**

db.employees.aggregate( [ { $match: { employee_ID : *'04f28c2a-f288-4194-accc-cfc1b585eee6'* } } ] )



# $redact stage

**Examples:**

```
db.employees.aggregate( [ { $redact : { $cond : [ { $in : [ 'Finance', '$acl' ] }, '$$DESCEND', '$$PRUNE'] } } ] )
db.employees.aggregate( [ { $redact : { $cond : [ { $in : ['Management', '$acl'] }, '$$DESCEND', '$$PRUNE'] } } ] )
db.employees.aggregate( [ { $redact : { $cond : [ { $in : ['HR', '$acl'] }, '$$DESCEND', '$$PRUNE'] } } ] )
```

# $out stage

- Takes the documents returned by the aggregation pipeline and writes them to a specified collection
- The $out stage must be the last stage in the pipeline. The $out operator lets the aggregation framework return result sets of any size

**Syntax**: { $out : { db : *<output-db>* , coll : *<output-collection>* } }

- The $out operation creates a new collection if one does not already exist.
- If the collection specified by the $out operation already exists, the $out stage atomically replaces the existing collection with the new results collection

**Example**:

```
db.movies.aggregate( [
                        { $group : { _id : '$year', count : { $sum : 1 } } },
                        { $sort : { count : -1 } },
                        { $out : { db : 'reporting' , coll : 'movies' } }
                    ] )
```

```
switched to db reporting
reporting> show collections
movies
reporting> db.movies.find()
[
  { _id: 1972, count: 338 },
  { _id: 1971, count: 333 },
  { _id: 1970, count: 311 },
  { _id: 1973, count: 303 },
  { _id: 1974, count: 301 },
  { _id: 1976, count: 284 },
  { _id: 1968, count: 281 },
  { _id: 1975, count: 278 },
  { _id: 1966, count: 266 },
  { _id: 1967, count: 265 },
  { _id: 1969, count: 259 },
  { _id: 1977, count: 249 },
  { _id: 1957, count: 232 },
  { _id: 1964, count: 221 },
```

# $merge stage

Writes the results of the aggregation pipeline to a specified collection. The $merge operator must be the last stage in the pipeline

- Can output to a collection in the same or different database.
- Creates a new collection if the output collection does not already exist
- Can incorporate results (insert new documents, merge documents, replace documents, keep existing documents, process documents with a custom update pipeline) into an existing collection.

**Syntax**:

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

# $merge stage

**Examples:**

```
db.movies.aggregate( [
                { $group : { _id : '$year', count : { $sum : 1 } } },
                { $sort : { count : -1 } },
                { $out : { db : 'reporti
            ] )


db.movies.aggregate( [
                { $group : { _id : '$ye
                { $sort : { count : -1 }
                { $merge : {
                        into : { db : 'repo
                        on : '_id',
                        whenMatched :
                        whenNotMatche
            ] )
```

```
{ _id: 1972, count: 338 },
{ _id: 1971, count: 333 },
{ _id: 1970, count: 311 },
{ _id: 1973, count: 303 },
```

```
{ _id: 1972, count: 338, title: 'Doomsday Machine' },
{ _id: 1971, count: 333, title: 'Isle of the Snake People' },
{ _id: 1970, count: 311, title: 'Kustom Kar Kommandos' },
{ _id: 1973, count: 303, title: 'The Death Wheelers' },
{ _id: 1974, count: 301, title: 'Out 1: Spectre' },
{ _id: 1976, count: 284, title: 'Chesty: A Tribute to a Legend' },
{ _id: 1968, count: 281, title: 'Tokugawa onna keibatsu-shi' },
{ _id: 1975, count: 278, title: 'Female Vampire' },
{ _id: 1966, count: 266, title: 'El Greco' },
{ _id: 1967, count: 265, title: 'Snow Devils' },
{ _id: 1969, count: 259, title: 'A Time for Dying' },
{ _id: 1977, count: 249, title: 'The Perfect Killer' },
{ _id: 1957, count: 232, title: 'A Hero of Our Times' },
{ _id: 1964, count: 221, title: 'The Human Dutch' },
{ _id: 1965, count: 217, title: 'Orgy of the Dead' },
```

# Views

- A MongoDB view is a queryable object whose contents are defined by an aggregation pipeline on other collections or views
- MongoDB does not persist the view contents to disk. A view's content is computed on-demand when a client queries the view
- **You can**:
  - ✓ Create a view on a collection of employee data to exclude any private or personal information (PII). Applications can query the view for employee data that does not contain any PII.
  - ✓ Create a view on a collection of collected sensor data to add computed fields and metrics. Applications can use simple find operations to query the data
  - ✓ …

# Create View

**Syntax**: db.createView( <viewName> , <source> , [<pipeline>] , <options> )

| Parameter | Type | Description |
|---|---|---|
| <viewName> | String | The name of the view to create. |
| <source> | String | The name of the source collection or view from which to create the view. You must create views in the same database as the source collection. |
| <pipeline> | Array | An array that consists of the aggregation pipeline stage(s). The view definition pipeline cannot include the $out or the $merge stage |
| <options> | Document | Optional. Additional options for the method. |

**Example**: create a view in aggregation DB

```
db.createView(
        'maleEmployees' ,
        'employees' ,
        [ { $match : { gender : 'male' } }, { $project : { acl : 0, employee_compensation : 0 } } ]
    )
```

Query the view: db.maleEmployees.find()

# Question?