B4M36DS2 – Database Systems 2

Lecture 9 – MongoDB. Aggregation and Indexing

27. 11. 2023

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

CourseWare Wiki   **https://cw.fel.cvut.cz/b231/courses/b4m36ds2/start**

Aggregation

- Aggregation pipeline

- Map-reduce function

- Single-purpose aggregation

Indexes

Aggregation

Aggregation operations collect values from documents, group them, and then perform different types of operations on that grouped data like sum, average, minimum, maximum, etc., to return a computed result.

MongoDB provides three ways to perform aggregation

- Aggregation pipeline

- Map-reduce function

- Single-purpose aggregation

Aggregation pipeline

Aggregation pipeline: Basics

- **Pipeline Stages**:
  - Aggregation operations use a pipeline, where each stage transforms the documents as they pass through the pipeline.

- **Order of Stages**:
  - The order of stages in the pipeline matters.

- **Immutability of Input Documents**:
  - Aggregation does not modify the original documents in the collection.

- **Memory Restrictions**:
  - Some stages, like $sort and $group have memory restrictions. MongoDB will produce an error if a stage exceeds 100MB of RAM.

- **Index Use**:
  - Some stages can take advantage of indexes ($match, $sort, etc.), greatly improving performance.

- **Output**:
  - The output of an aggregation query is a cursor to the documents that match the pipeline of operations.
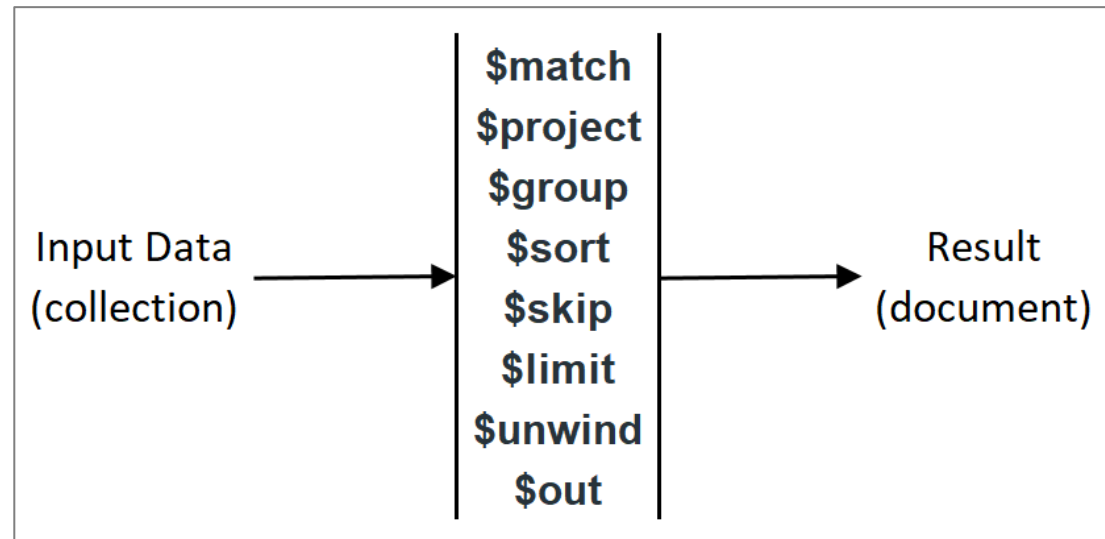
Aggregation pipeline

An **aggregation pipeline** consists of one or more **stages** that process documents:

Each stage **operates** on the input documents. The documents' output from one stage is passed to the next stage.

An aggregation pipeline can return results for groups of documents.

Aggregation stages

- Filtering

- Sorting

- Grouping

- String concatenation

- Array processing

- etc.

Aggregation pipeline

An aggregation pipeline is constructed with the **aggregate()** method of the **Collection** class and takes an array of stages:

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

Stages can appear multiple times in a pipeline except for $**out**, $**merge**, and $**geoNear**.

The **order** of the stages can significantly affect the performance of the query.

$**merge** or $**out** stages modify documents.

Data set for this lecture

```
db.orders.insertMany( [

        { _id: 0, name: "Pepperoni", size: "small", price: 19,  quantity: 10 },

        { _id: 1, name: "Pepperoni", size: "medium", price: 20,  quantity: 20},

        { _id: 2, name: "Pepperoni", size: "large", price: 21, quantity: 30},

        { _id: 3, name: "Cheese", size: "small", price: 12, quantity: 15 },

        { _id: 4, name: "Cheese", size: "medium", price: 13, quantity: 50 },

        { _id: 5, name: "Cheese", size: "large", price: 14, quantity: 10},

        { _id: 6, name: "Vegan", size: "small", price: 17, quantity: 10},

        { _id: 7, name: "Vegan", size: "medium", price: 18, quantity: 10}

   ] )
```

Source: https://www.mongodb.com

db.**orders**.aggregate([

Collection

// Stage 1: Filter pizza order documents by pizza size

{

$match stage $match: { size: **"medium"** }

},

// Stage 2: Group remaining documents by pizza name and calculate total quantity

{

$group stage $group: { _id: **"$name"**, totalQuantity: { $sum: **"$quantity"** } }

}

])

Stage   Expression   Accumulator

```
[
    { _id: 'Cheese', totalQuantity: 50 },
    { _id: 'Vegan', totalQuantity: 10 },
    { _id: 'Pepperoni', totalQuantity: 20 }
]
```

$match and $group stages

```
db.orders.aggregate( [
        {$match: { size: "medium" }},
        {$group: { _id: "$name", totalQuantity: { $sum: "$quantity" } } }
])
```

Intermediate result of the $match (filtering stage):

```
{ _id: 1, name: "Pepperoni", size: "medium", price: 20, quantity: 20},
{ _id: 4, name: "Cheese", size: "medium", price: 13, quantity:50},
{ _id: 7, name: "Vegan", size: "medium", price: 18, quantity: 10}
```

Final result of the query

```
[
    { _id: 'Cheese', totalQuantity: 50 },
    { _id: 'Vegan', totalQuantity: 10 },
    { _id: 'Pepperoni', totalQuantity: 20 }
]
```

$match and $group stages

- **$match stage** is used for **filtering** the documents.

  - It can reduce the amount of documents given as input to the next stage.

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

- **$group stage** is used to **group** documents based on some value.

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

$group does *not* order its output documents.

Aggregation pipeline: Accumulators

Accumulators are used in the group stage.

- **sum**
  - Sums numeric values for the documents in each group

- **count**
  - Counts the total number of documents

- **avg**
  - Calculates the average of all given values from all documents

- **min / max**
  - It gets the minimum / maximum value from all the documents

- **first / last**
  - Gets the first / last document from the grouping

Aggregation pipeline: Accumulators

The total number of all pizza types and maximum quantity for each type

```
db.orders.aggregate([

    {

        $group: { _id: "$name",

                 total_offers: { $sum: 1 },

                 max_quantity: { $max: "$quantity" }}

    }

])
```

> **Result**
> ```
> [
>     { _id: 'Pepperoni', total_offers: 3, max_quantity: 30 },
>     { _id: 'Cheese', total_offers: 3, max_quantity: 50 },
>     { _id: 'Vegan', total_offers: 2, max_quantity: 10 }
> ]
> ```

Expressions refer to the name of the field in input documents, e.g.

{ **$group** : { _id : "**$name**", total: { $sum: "**$quantity**" }}}

Example: average price for each pizza size
```
db.orders.aggregate([
    {
        $group : {
            _id : "$size",
            count: { $sum: 1 },
            averagePrice: { $avg: "$price" }
        }
} ])
```

```
[
  { _id: 'large', count: 2, averagePrice: 17.5 },
  { _id: 'small', count: 3, averagePrice: 16 },
  { _id: 'medium', count: 3, averagePrice: 15 }
]
```

`$count stage` passes a document to the next stage that contains a count of the number of documents input to the stage.

<div align="center">

`{ $count: <string> }`

</div>

Example: Display the total number of offers for Pepperoni pizzas

db.orders.aggregate([

    { **$match**: { name: "Pepperoni" } },

    { **$count**: "Total offers for Pepperoni" }

])

> **Result**
> [{ 'Total offers for Pepperoni': 3 }]

Displaying the total number of pizzas with price >= 20

```
db.orders.aggregate([

    { $match: { price: { $gte: 20 } } },

    { $count: "Total number of expensive pizzas" }

])
```

Result
[{ 'Total number of expensive pizzas': 2 }]

Aggregation pipeline: Example

Count the total quantity and average price of pizzas of each type that have a size of "medium" or "large"

```
db.orders.aggregate([
    {

        $match: { size: { $in: ["medium", "large"] } }
    },
    {

        $group : {
            _id : "$name",
            totalQuantity: { $sum: "$quantity" },
            averagePrice: { $avg: "$price" }
        }
    }
])
```

```
[
  { _id: 'Pepperoni', totalQuantity: 50, averagePrice: 20.5 },
  { _id: 'Cheese', totalQuantity: 60, averagePrice: 13.5 },
  { _id: 'Vegan', totalQuantity: 10, averagePrice: 18 }
]
```

Aggregation pipeline: $sort, $limit and $skip stages

- **$sort stage** is used to sort the documents, that is, rearrange them.

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

  - **Sort order:** 1 – sort ascending, -1 – sort descending

- **$limit stage** is used to pass the first **n** number of documents, thus limiting them.

```
{ $limit: <positive 64-bit integer> }
```

- **$skip stage** is used to skip **n** number of documents and pass the remaining documents.

```
{ $skip: <positive 64-bit integer> }
```

Example: Display the three most expensive pizzas sorted by price in descending order

```
db.orders.aggregate([

    { $sort: { price: -1 } },

    { $limit: 3 }

])
```

```
[
  {
    _id: 2,
    name: 'Pepperoni',
    size: 'large',
    price: 21,
    quantity: 30,
    date: ISODate("2021-03-17T09:22:12.000Z")
  },
  {
    _id: 1,
    name: 'Pepperoni',
    size: 'medium',
    price: 20,
    quantity: 20,
    date: ISODate("2021-03-13T09:13:24.000Z")
  },
  {
    _id: 0,
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    quantity: 10,
    date: ISODate("2021-03-13T08:14:30.000Z")
  }
]
```

Aggregation pipeline: Example

Count the total quantity and average price of pizzas of each type that have a size of "small" or a price less than 15, and sort the results by descending average price:

```
db.orders.aggregate([
    {
        $match: { $or: [ { size: "small" }, { price: { $lt: 15 } } ] }
    },
    {
        $group : {
            _id : "$name",
            totalQuantity: { $sum: "$quantity" },
            averagePrice: { $avg: "$price" }
        }
    },
    {
        $sort: { averagePrice: -1 }
    }
])
```

```
[
  { _id: 'Pepperoni', totalQuantity: 10, averagePrice: 19 },
  { _id: 'Vegan', totalQuantity: 10, averagePrice: 17 },
  { _id: 'Cheese', totalQuantity: 75, averagePrice: 13 }
]
```

$project

Reshapes each document in the stream, such as by adding new fields or removing existing fields, moving and renaming fields, and building computed fields.

For each input document, outputs one document.

{ **$project** : { name : 1, price : 1 } }

Only pass on fields "name" and "price"

$addFields

Adds new fields to documents. $addFields outputs documents that contain all existing fields from the input documents and newly added fields.

The **$addFields** stage is equivalent to a **$project** stage that explicitly specifies all existing fields in the input documents and adds the new fields.

Aggregation pipeline: Example

Find the name and size of the most expensive pizza:

```
db.orders.aggregate([
    {
        $sort: { price: -1 }
    },
    {
        $limit: 1
    },
    {
        $project: { name: 1, size: 1, _id: 0 }
    }
])
```

[{ name: 'Pepperoni', size: 'large' }]

Aggregation pipeline: Example

Count the total revenue from selling pizzas of each type

```
db.orders.aggregate([
    {
        $project: { name: 1, revenue: { $multiply: ["$price", "$quantity"] } }
    },
    {
        $group : {
            _id : "$name",
            totalRevenue: { $sum: "$revenue" }
        }
    }
])
```

```
[
  { _id: 'Pepperoni', totalRevenue: 1220 },
  { _id: 'Vegan', totalRevenue: 350 },
  { _id: 'Cheese', totalRevenue: 970 }
]
```

Aggregation pipeline: Example

Find the name, size, and price of the first three pizzas that have a quantity of 10 or more, and add a field called "discount" that shows the percentage of the price reduction if the quantity is increased by 10

```
db.orders.aggregate([
    {
        $match: { quantity: { $gte: 10 } }
    },
    {
        $limit: 3
    },
    {
        $project: { name: 1, size: 1, price: 1,
        discount: { $multiply: [ { $divide: [ { $subtract: [ "$price",
        { $multiply: [ "$price", 0.9 ] } ] }, "$price" ] }, 100 ] } }
    }
])
```

```
[
  {
    _id: 0,
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    discount: 9.999999999993
  },
  {
    _id: 1,
    name: 'Pepperoni',
    size: 'medium',
    price: 20,
    discount: 10
  },
  {
    _id: 2,
    name: 'Pepperoni',
    size: 'large',
    price: 21,
    discount: 9.99999999999
  }
]
```

- No built-in way to **format** floating point numbers
- **$round** operator
- **$trunc** operator

```
db.orders.aggregate([
    {

        $match: { quantity: { $gte: 10 } }

    },
    {

        $limit: 3

    },
    {

        $project: { name: 1, size: 1, price: 1,
        discount: {$round: { $multiply: [ { $divide: [ { $subtract:
        [ "$price", { $multiply: [ "$price", 0.9 ] } ] }, "$price" ] },
        100 ] } } }

    }
```

```
[
  {
    _id: 0,
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    discount: 10
  },
  {
    _id: 1,
    name: 'Pepperoni',
    size: 'medium',
    price: 20,
    discount: 10
  },
  {
    _id: 2,
    name: 'Pepperoni',
    size: 'large',
    price: 21,
    discount: 10
  }
]
```

Add a new field with total price of all pizzas.

```
db.orders.aggregate([
    {
        $addFields:
            {
                "totalPrice": { $multiply: [ "$price", "$quantity" ] }
            }
    }
])
```

```
[
 {
  _id: 0,
  name: 'Pepperoni',
  size: 'small',
  price: 19,
  quantity: 10,
  totalPrice: 190
 },
 {
  _id: 1,
  name: 'Pepperoni',
  size: 'medium',
  price: 20,
  quantity: 20,
  totalPrice: 400
 },
 …
```

Aggregation pipeline: $unset stage

- **$unset stage** removes/excludes fields from documents.

To remove a single field:
```
{ $unset: "<field>" }
```

To remove multiple fields:
```
{ $unset: [ "<field1>", "<field2>", ... ] }
```

To remove/exclude a field or fields within an embedded document:
```
{ $unset: "<field.nestedfield>" }
{ $unset: [ "<field1.nestedfield>", ...] }
```

The $unset is an alias for the $project stage that removes/excludes fields:
```
{ $project: { "<field1>": 0, "<field2>": 0, ... } }
```

$replaceRoot and $replaceWith stages

- **$replaceRoot stage** replaces the input document with the specified document.
  - The operation replaces all existing fields in the input document, including the _id field.

  `{ $replaceRoot: { newRoot: <replacementDocument> } }`

- **$replaceWith stage** peforms the same action as the **$replaceRoot** stage, but the stages have different forms.

  `{ $replaceWith: <replacementDocument> }`

  The **$replaceRoot** and **$replaceWith** stages require the new root to be an object

```
db.orders.aggregate([
    {
        $replaceRoot: { newRoot: { "size": "$size", "name": "$name" } }
    }
])

db.orders.aggregate([
    {
        $replaceWith: { "size": "$size", "name": "$name" }
    }
])
```

The $replaceRoot and $replaceWith stages require the new root to be an object.

```
[
  { size: 'small', name: 'Pepperoni' },
  { size: 'medium', name: 'Pepperoni' },
  { size: 'large', name: 'Pepperoni' },
  { size: 'small', name: 'Cheese' },
  { size: 'medium', name: 'Cheese' },
  { size: 'large', name: 'Cheese' },
  { size: 'small', name: 'Vegan' },
  { size: 'medium', name: 'Vegan' }
]
```

$mergeObjects

- **$mergeObjects** combines multiple documents into a single document.

```
{ $mergeObjects: <document> }
```

- Ignores null operands.
- Overwrites the field values as it merges the documents.

```
db.orders.aggregate([
    {
        $addFields: {
            mergedField: {
                $mergeObjects: { "size": "$size", "name": "$name" }
            }
        }
    }
])
```

```
[
  {
    _id: 0,
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    quantity: 10,
    mergedField: { size: 'small', name: 'Pepperoni' }
  },
  ...
```

Define bestsellers of each pizza type

```
db.orders.aggregate(
{
    $addFields: { "total": { $multiply: [ "$price", "$quantity" ] } }
},
{
    $sort: { "total": -1 }
},
{
    $group: { _id: "$name", "documents": { $push: "$$ROOT" } }
},
{
    $replaceRoot: { newRoot: { $arrayElemAt: [ "$documents", 0 ] } }
},
{ $unset: "total" },
{
    $sort: {"name": 1 }
}
)
```

Create the **total** field as a product of the price and quantity. Use this field for sorting

Group pizzas and create array **documents** with pizzas details

```
[
  { _id: 4, name: 'Cheese', size: 'medium', price: 13, quantity: 50 },
  { _id: 2, name: 'Pepperoni', size: 'large', price: 21, quantity: 30 },
  { _id: 7, name: 'Vegan', size: 'medium', price: 18, quantity: 10 }
]
```

Define bestsellers of each pizza type

```
db.orders.aggregate(
    {
        $addFields: { "total": { $multiply: [ "$price", "$quantity" ] } }
    },
    {
        $sort: { "total": -1 }
    },
    {
        $group: { _id: "$name" }
    }
)
```

```
[ { _id: 'Pepperoni' }, { _id: 'Vegan' }, { _id: 'Cheese' } ]
```

$out and $merge stages

- **$out** takes the documents returned by the aggregation pipeline and writes them to a specified collection.

```
{ $out: { db: "<output-db>", coll: "<output-collection>" } }
```

```
{ $out: "<output-collection>" } // Output collection is in the same databas
```

- **$merge** writes the results of the aggregation pipeline to a specified collection.

```
{$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
}}
```

Aggregation pipeline: Example

Find the **name and size** of the pizzas that have the **lowest price**,
and **sort** the results by descending name

```
var minPrice = db.orders.aggregate([
{
        $group:
        {
                _id: null,
                minPrice: { $min: "$price" }
        }
}
] ).toArray()[0].minPrice;

db.orders.find({ price: minPrice })
```

Trade-off: easy to understand, but it might be less efficient because it requires two separate database operations

Aggregation result:

[{ _id: null, minPrice: **13** }]

```
[
  { _id: 4, name: 'Cheese', size: 'medium', price: 13, quantity: 50 }
]
```

Find the name and size of the pizzas that have the lowest price, and sort the results by descending name

Trade-off: This method is more efficient because it only requires one database operation, but it might be harder to understand separate database operations

```
db.orders.aggregate([
    { $project: { name: 1, size: 1, price: 1 } },
    { $sort: { price: 1 } },
    { $group: { _id: null, minPrice: { $min: "$price" },
                pizzas: { $push: { name: "$name", size: "$size", price: "$price" } } } },
    { $unwind: "$pizzas" },
    { $redact: {
        $cond: {
            if: { $eq: [ "$pizzas.price", "$minPrice" ] },
            then: "$$KEEP",
            else: "$$PRUNE" }
        }
     },
    { $project: { name: "$pizzas.name", size: "$pizzas.size", _id: 0 } },
    { $sort: { name: -1 } }
])
```

[{ name: 'Cheese', size: 'small' }]

Aggregation pipeline: $lookup stage

- **$lookup** Performs a left outer join to a collection in the *same* database to filter in documents from the "joined" collection for processing.
    - The $lookup stage adds a new array field to each input document.
    - The new array field contains the matching documents from the "joined" collection.
    - The $lookup stage passes these reshaped documents to the next stage.

```
{

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

Aggregation pipeline: $lookup stage

```
db.customers.insertMany( [
      { _id: 100, name: "Anna", address: "123 Main St" },
      { _id: 101, name: "Matej", address: "456 Maple Ave" },
      { _id: 102, name: "Tomas", address: "789 Oak Dr" },
      { _id: 103, name: "Helena", address: "321 Pine Ln" }
] )

db.orders.find();
[
 { _id: 0, name: 'Pepperoni', size: 'small', price: 19, quantity: 10, customerId: 100},
 { _id: 1, name: 'Pepperoni', size: 'medium', price: 20, quantity: 20, customerId: 100},
 { _id: 2, name: 'Pepperoni', size: 'large', price: 21, quantity: 30, customerId: 101 },
 { _id: 3, name: 'Cheese', size: 'small', price: 12, quantity: 15, customerId: 101 },
 { _id: 4, name: 'Cheese', size: 'medium', price: 13, quantity: 50, customerId: 102 },
 { _id: 5, name: 'Cheese', size: 'large', price: 14, quantity: 10, customerId: 102 },
 { _id: 6, name: 'Vegan', size: 'small', price: 17, quantity: 10, customerId: 103 },
 { _id: 7, name: 'Vegan', size: 'medium', price: 18, quantity: 10, customerId: 103 }
]
```

Define the details of the orders placed by Tomas, including the name and size of the pizza, the price, and the quantity ordered.

```
db.orders.aggregate([
    {

        $lookup:

            {

                from: "customers",
                localField: "customerId",
                foreignField: "_id",
                as: "customer_info"

            }

    },
    {

        $match: { "customer_info.name": "Tomas" }

    }

])
```

```
[
  {
    _id: 4,
    name: 'Cheese',
    size: 'medium',
    price: 13,
    quantity: 50,
    customerId: 102,
    customer_info: [ { _id: 102, name:
'Tomas', address: '789 Oak Dr' } ]
  },
  {
    _id: 5,
    name: 'Cheese',
    size: 'large',
    price: 14,
    quantity: 10,
    customerId: 102,
    customer_info: [ { _id: 102, name:
'Tomas', address: '789 Oak Dr' } ]
  }
]
```

Who ordered the Pepperoni pizza?

```
db.orders.aggregate([
    {
        $match: { name: { $regex: /^Pepperoni$/i } }
    },
    {
        $lookup:
            {
                from: "customers",
                localField: "customerId",
                foreignField: "_id",
                as: "customer_info"
            }
    }
])
```

```
[
  {
    _id: 0,
    name: 'Pepperoni',
    size: 'small',
    price: 19,
    quantity: 10,
    customerId: 100,
    customer_info: [ { _id: 100, name:
'Anna', address: '123 Main St' } ]
  },
  {
    _id: 1,
    name: 'Pepperoni',
    size: 'medium',
    price: 20,
    quantity: 20,
    customerId: 100,
    customer_info: [ { _id: 100, name:
'Anna', address: '123 Main St' } ]
  },
  {
    _id: 2,
    name: 'Pepperoni',
    size: 'large',
    price: 21,
    quantity: 30,
    customerId: 101,
    customer_info: [ { _id: 101, name:
'Matej', address: '456 Maple Ave' } ]
  }
]
```

SQL to Aggregation Mapping Chart

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum $sortByCount
JOIN	$lookup
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge
UNION ALL	$unionWith

MapReduce

MapReduce

MapReduce is used for aggregating results for a large volume of data. Map reduce has two main functions one is a **map** that groups all the documents, and the second one is the **reduce** which performs operations on the grouped data.
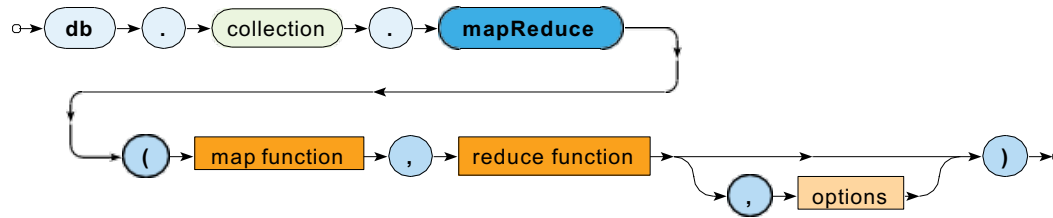
Syntax:

> db.collectionName.**mapReduce**(
> > mappingFunction,
> > reduceFunction,
> > { out : 'Result' }
>
>)

Starting in MongoDB 5.0, map-reduce is deprecated

Read more: Map-Reduce to Aggregation Pipeline
https://www.mongodb.com/docs/manual/reference/map-reduce-to-aggregation-pipeline/

MapReduce

Executes a **MapReduce** job on a selected collection

- Parameters
  - **Map**: JavaScript implementation of the Map function
  - **Reduce**: JavaScript implementation of the Reduce function
  - **Options**

MapReduce

Map function

- Current document is accessible via `this`
- `emit(key, value)` is used for emissions

Reduce function

- Intermediate key and values are provided as arguments
- Reduced value is published via `return`

Options

- `query`: only matching documents are considered
- `sort`: they are processed in a specific order
- `limit`: at most a given number of them is processed
- `out`: output is stored into a given collection

Define the total quantity for each pizza type

```
var mapfunction = function()
        {emit(this.name, this.quantity)}


var reducefunction = function(key, values)
        {return Array.sum(values)}


db.orders.mapReduce(mapfunction, reducefunction, {'out':'Result'})


db.Result.find()
[
 { _id: 'Vegan', value: 20 },
 { _id: 'Pepperoni', value: 60 },
 { _id: 'Cheese', value: 75 }
]
```

Define the total quantity for each pizza type

```
db.orders.aggregate([
{
    $group :
        {
            _id : "$name",
            totalQuantity: { $sum: "$quantity" }
        }
}
])
```

```
[
  { _id: 'Vegan', value: 20 },
  { _id: 'Pepperoni', value: 60 },
  { _id: 'Cheese', value: 75 }
]
```

Define the total price for each pizza type

```
var mapFunction = function() {
    emit(this.name, this.price * this.quantity);
};


var reduceFunction = function(key, values) {
    return Array.sum(values);
};


db.orders.mapReduce(
    mapFunction,
    reduceFunction,
    { out: "totalCostPerPizzaType" }
)
```

```
[
  { _id: 'Cheese', totalCost: 1440 },
  { _id: 'Vegan', totalCost: 350 },
  { _id: 'Pepperoni', totalCost: 1220 }
]
```

Define the total cost for each pizza type

```
db.orders.aggregate([
    {
        $group :
            {
                _id : "$name",
                totalCost: { $sum: { $multiply: [ "$price", "$quantity" ] } } }
            }
    }
])
```

```
[
  { _id: 'Pepperoni', totalCost: 1220 },
  { _id: 'Vegan', totalCost: 350 },
  { _id: 'Cheese', totalCost: 1440 }
]
```

Single Purpose Aggregation

Single Purpose Aggregation

It is used when we need simple access to document like counting the number of documents or for finding all distinct values in a document. It simply provides the access to the common aggregation process using the **count()**, **distinct()**, and **estimatedDocumentCount()** methods, so due to which it lacks the flexibility and capabilities of the pipeline.

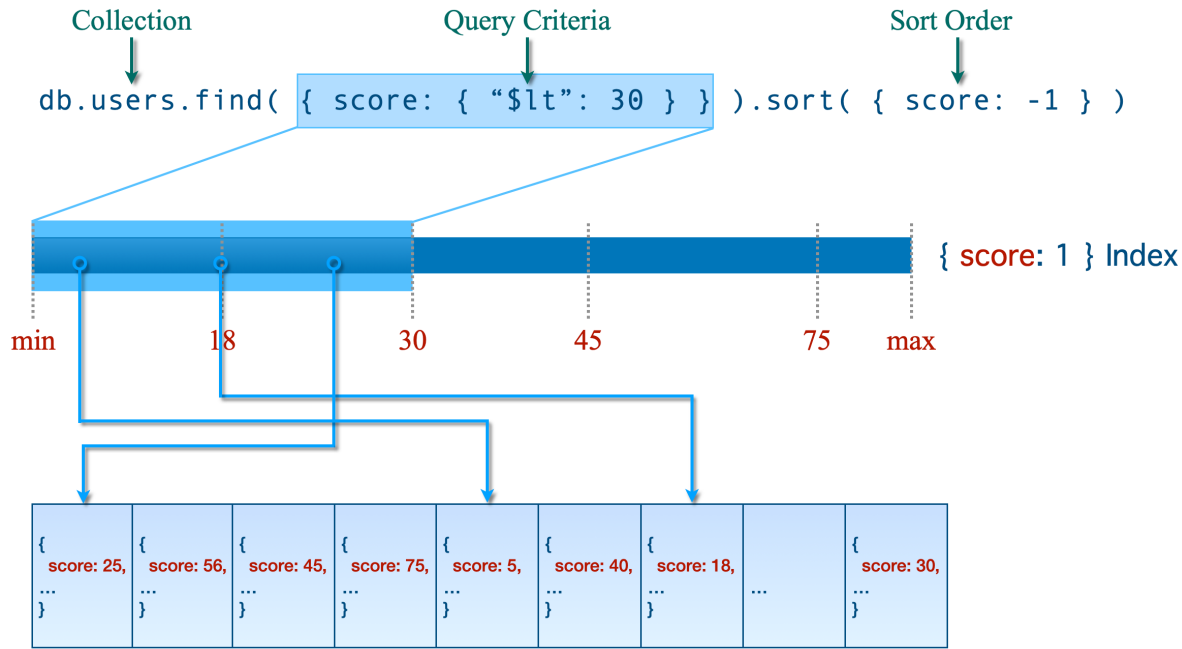db.orders.**distinct**("name")
['Cheese', 'Pepperoni', 'Vegan']

db.orders.find({ name: "Pepperoni" }).**count**()
3

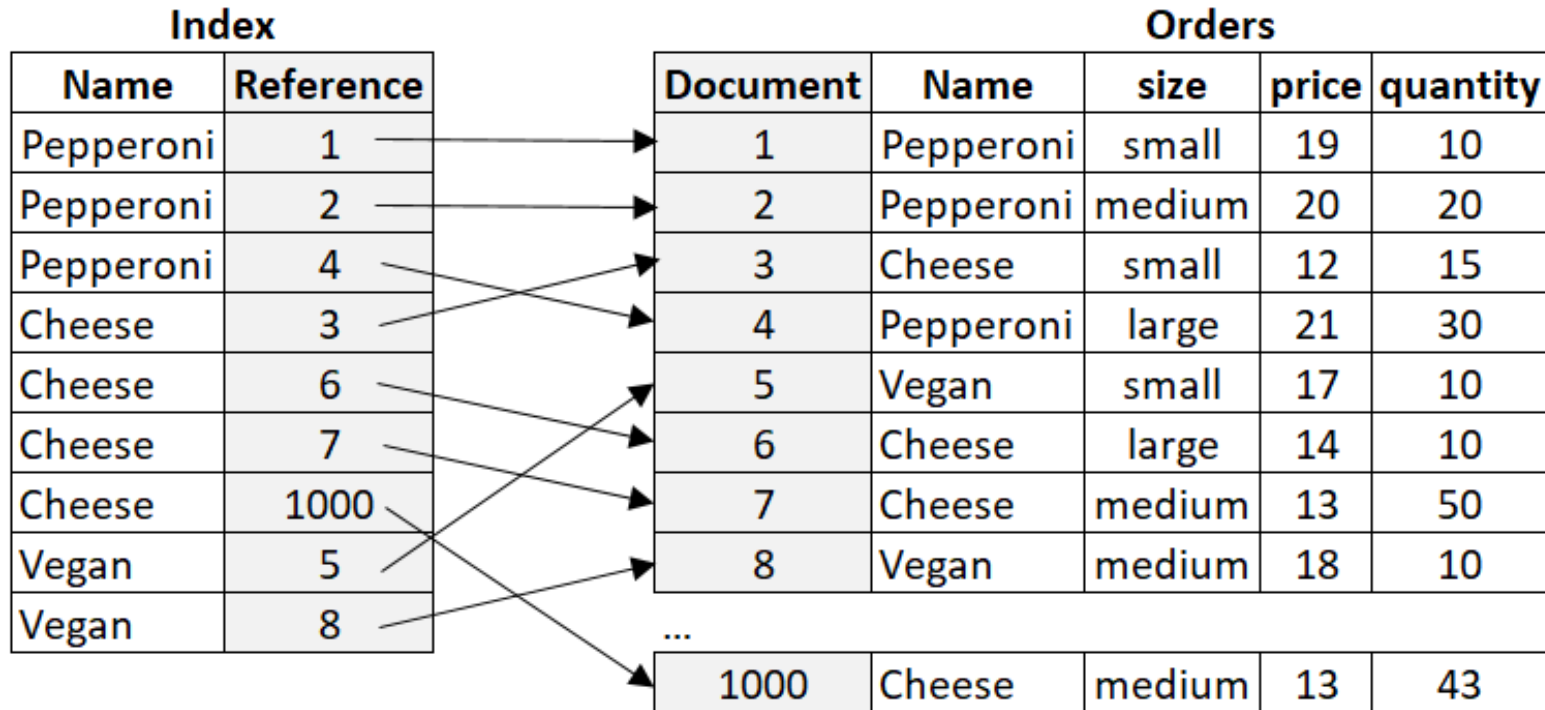db.orders.**estimatedDocumentCount**()
8

Indexing

Indexes

- Speedup queries
- MongoDB uses B-Tree indexes
- Can build the index on any field of the document
- Skips documents that do not have the indexed field (Sparse index)

Collection   Query Criteria   Sort Order

db.users.find({ score: { "$lt": 30 } }).sort({ score: -1 })

{ score: 1 } Index

min   18   30   45   75   max

| { score: 25, ... } | { score: 56, ... } | { score: 45, ... } | { score: 75, ... } | { score: 5, ... } | { score: 40, ... } | { score: 18, ... } | ... | { score: 30, ... } |

https://www.cloudduggu.com/mongodb/indexing/   users

Indexes

Index

Name	Reference
Pepperoni	1
Pepperoni	2
Pepperoni	4
Cheese	3
Cheese	6
Cheese	7
Cheese	1000
Vegan	5
Vegan	8

Orders

Document	Name	size	price	quantity
1	Pepperoni	small	19	10
2	Pepperoni	medium	20	20
3	Cheese	small	12	15
4	Pepperoni	large	21	30
5	Vegan	small	17	10
6	Cheese	large	14	10
7	Cheese	medium	13	50
8	Vegan	medium	18	10
...				
1000	Cheese	medium	13	43

Indexes: Good Practices

Good Indexing Practices

- **Use Compound Indexes**:

    - Compound indexes are indexes composed of several different fields.

- **Follow the ESR rule**:
    - First, add those fields against which Equality queries are run.
    - The next fields to be indexed should reflect the Sort order of the query.
    - The last fields represent the Range of data to be accessed.

- **Use Covered Queries When Possible**:
    - Covered queries return results from an index directly without accessing the source documents.

Indexes: Bad Practices

Bad Indexing Practices

- **Over-indexing**:
  - Adding an index has a negative performance impact for write operations. For collections with a high write-to-read ratio, indexes are expensive because each insert must update indexes.

- **Indexing Rarely Queried Fields** :
  - Additional fields also increase the size of the index data, which can be detrimental if those fields are not frequently queried.

- **Not Considering the Working Set** :
  - The working set is the portion of the data and indexes that fit in RAM. If your indexes do not fit into RAM, MongoDB must read the index data from disk, which is much slower than reading from RAM.

db.collection.**createIndex**(

 <key and index type specification>,

 <options>

)


```
db.orders.createIndex( { name: -1 } )
```

The default **name** for an index is the concatenation of the indexed keys and each key's direction in the index (i.e. 1 or -1) using underscores as a separator.
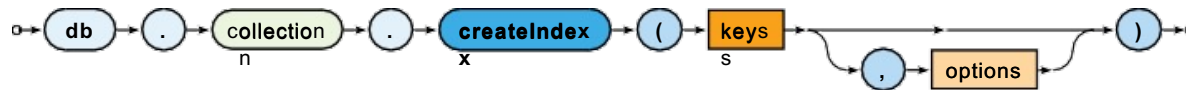
`{ item : 1, quantity: -1 }` has the name `item_1_quantity_-1`

You can create indexes with a custom name, such as one that is more human-readable than the default
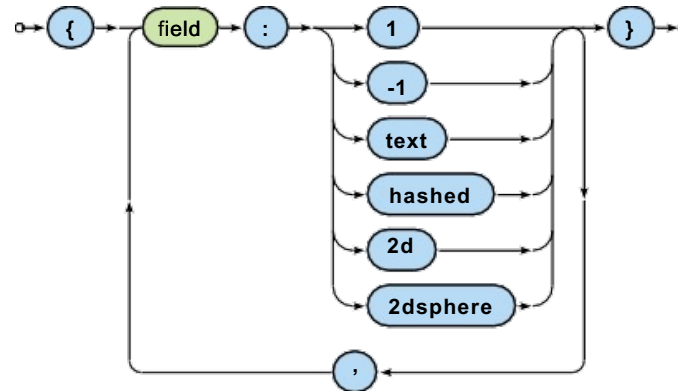
db.orders.createIndex(

   { name: 1, size: -1 },

   { name: **"query for size"** }

)

db.collection.getIndexes() method – view index names

Secondary **index creation**

Definition of keys (fields) to be involved

Indexes: Index Types

- Single Field

- Compound Index

- Multikey Index

- Geospatial Index

- Text Indexes

- Hashed Indexes

- Clustered Indexes

Index Structures

Index forms

- One key / multiple keys (**composed index**)
- Ordinary fields / array fields (**multi-key index**)

Index properties

- **Unique** – duplicate values are rejected (cannot be inserted)
- **Partial** – only certain documents are indexed
- **Sparse** – documents without a given field are ignored
- **TTL** – documents are removed when a timeout elapses

Just some type / form / property combinations can be used!

Aggregation

- Aggregation pipeline. Stages, accumulators, expressions
- Map-reduce function
- Single-purpose aggregation

Indexes

Resources

https://www.mongodb.com/docs/manual/aggregation/

https://www.mongodb.com/docs/manual/indexes/

https://www.geeksforgeeks.org/aggregation-in-mongodb