NoSQL Databases

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Data aggregation examples

Data Model

- Given the following collection of books

```
{
 "title":"MongoDb Guide2",
 "tag":["mongodb","guide","database"],
 "n":200,
 "review_score": 2.2,
 "price":[ {"v": 22.22, "c": "€", "country": "IT"},
           {"v": 22.00, "c": "£", "country": "UK"}
         ],
 "author": {
    "_id": 1,
    "name":"Mario",
    "surname": "Rossi"}
}
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:"Developing with Python",
 tag:["python","guide","programming"],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: "€",  country: "IT"},
        {v: 19.49, c: "£", country:"UK"} ],
 author: {_id: 2,
          name:"John",
          surname: "Black"}
}, …
```

price currency

price value

number of pages

Example 1

- For each country, select the average price and the average review_score.
- The review score should be rounded down.
- Show the first 20 results with a total number of books higher than 50.

$unwind

```
db.book.aggregate( [
  { $unwind: "$price" } ,
] )
```

Build a document for each entry of the **price** array

Result - `$unwind`

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 19.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 18, "c" : "£", "country" : "UK" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 24.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 19.49, "c" : "£", "country" : "UK" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

…

$group

```
db.book.aggregate( [
  { $unwind: "$price" } ,
  { $group: { _id: "$price.country"},
              avg_price: { $avg: " $price.v" ,
              bookcount: {$sum:1},
              review: {$avg: " $review_score"}
            }
  }
] )
```

dot notation to access the value of the **embedded** document fields

count the number of books (number of documents)

Result - **$group**

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}
{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}
{ "_id" : "US", "avg_price" : 22.49, "bookcount": 49, "review": 4.2}
…
```

$match

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
              avg_price: { $avg: '$price.v' },
              bookcount: {$sum:1},
              review: {$avg: '$review_score'}
            }
  },
 {$match: { bookcount: { $gte: 50 } } },
] )
```

Filter the documents where **bookcount** is greater than 50

Result - **`$match`**

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}

{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}

…
```

$project

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
] )
```

round down the review score

Result - `$project`

```
{ "_id" : "UK", "avg_price" : 18.75, "review": 4}
{ "_id" : "IT", "avg_price" : 22.49, "review" :  3}
…
```

$limit

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
              avg_price: { $avg: '$price.v' },
              bookcount: {$sum:1},
              review: {$avg: '$review_score'}
            }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
 {$limit:20}
] )
```

Limit the results to the first 20 documents

Example 2

- Compute the 95 percentile of the number of pages,
- only for the books that contain the tag “guide”.

$match

```
db.book.aggregate( [
  {$match: { tag : "guide"} }
] )
```

select documents containing “guide” in the tag array, compare with tag:[“guide”]

Result - $match

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

…

$sort

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
 {$sort : { n: 1} }
] )
```

sort the documents in ascending order according to the value of the **n** field, which stores the number of pages of each book

Result - `$sort`

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], **"n" : 100**, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], **"n" : 352**, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

…

$group + $push

```
db.book.aggregate( [
  {$match: { tag : "guide"} },
  {$sort : { n: 1} },
  {$group: {_id:null, value: {$push: "$n"}}}
] )
```

group all the records together inside a single document (**_id:null**), which contains an array with all the values of **n** of all the records

Result - `$group + $push`

```
{ "_id": null, "value": [100, 352, …]}
```

$project + $arrayElemAt

```
db.book.aggregate( [
  {$match: { tag : "guide"} },
  {$sort : { n: 1} },
  {$group: {_id:null, value: {$push: "$n"}}},
  {$project:
            {"n95p": {$arrayElemAt:
                      ["$value",
                      {$floor: {$multiply: [0.95, {$size: "$value"}]}}
                     ]
            } }
}
] )
```

get the value of the array at a given index with { $arrayElemAt: [<array>, <idx>] }

compute the index at 95% of the array length

Result - **`$project + $arrayElemAt`**

```
{ "_id" : null, "n95p" : 420 }
```

Example 3

- Compute the median of the review_score,
- only for the books having at least a price whose value is higher than 20.0.

Solution

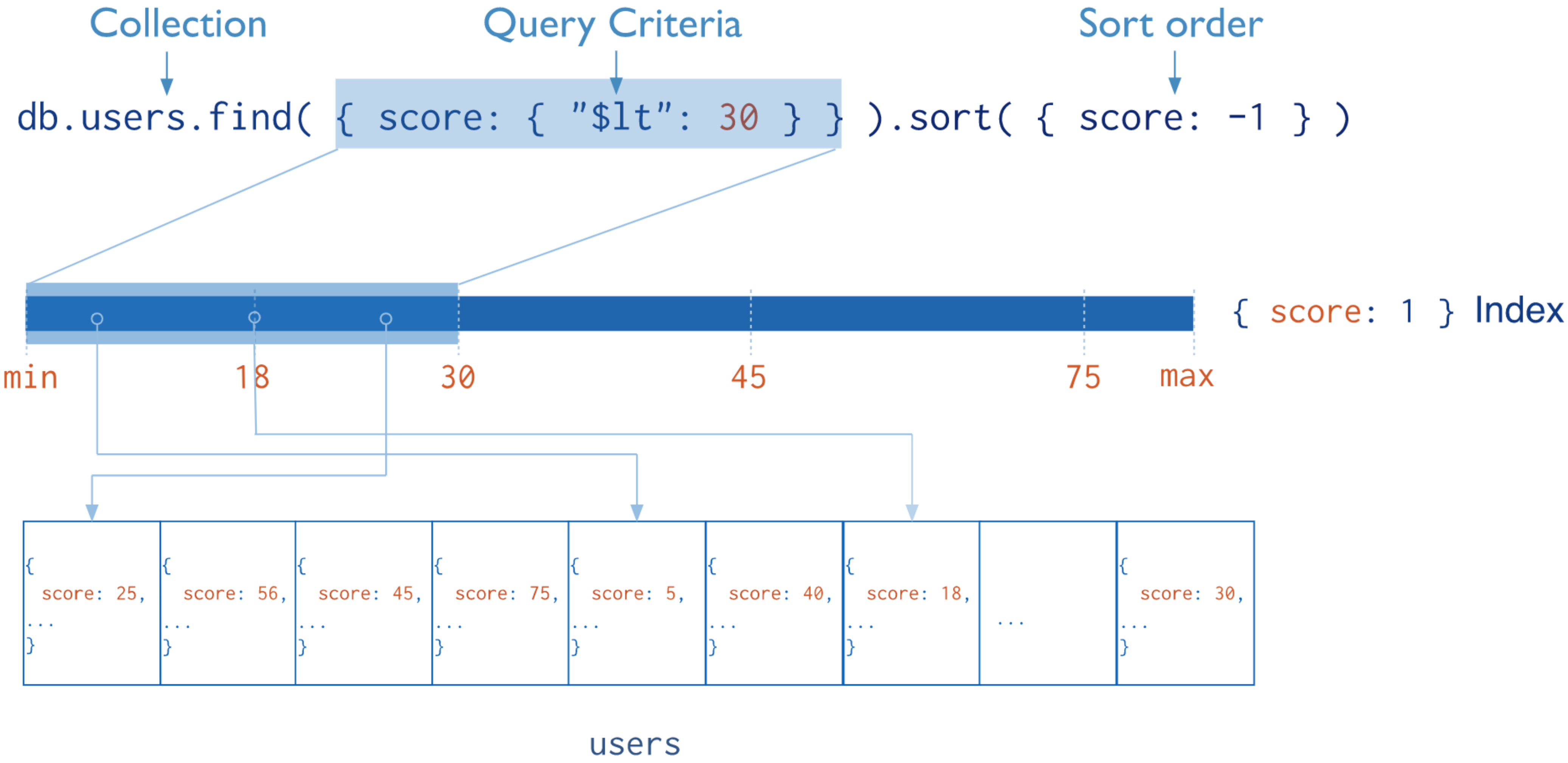
```
db.book.aggregate( [
  {$match: {'price.v' : { $gt: 20 }} },
  {$sort : {review_score: 1} },
  {$group: {_id:null, rsList: {$push: '$review_score'}}},
  {$project:
        {'median': {$arrayElemAt:
                  ['$rsList',
                  {$floor: {$multiply: [0.5, {$size: '$rsList'}]}}
              ]
        } }
}
] )
```

Indexing

Indexes

- Without indexes, MongoDB must perform a **collection scan**, i.e. scan every document in a collection, to select those documents that match the query statement.
- Indexes are data structures that store a small portion of the collection's data set in a form easy to traverse.
- They store **ordered values of a specific field**, or set of fields, in order to efficiently support
    - equality matches,
    - range-based queries and
    - sorting operations.

Indexes


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,
...
}
users

Indexes

- MongoDB creates a unique index on the **_id** field during the creation of a collection.
- The **_id** index prevents clients from inserting two documents with the same value for the **_id** field.
- You cannot drop this index on the **_id** field.

Create new indexes

- Creating an index

```
db.collection.createIndex(<index keys>, <options>)
```

  - Before v. 3.0 use `db.collection.ensureIndex()`

- Options include:
  - `name` - a mnemonic name given by the user, you cannot rename an index once created, instead, you must drop and re-create the index with a new name
  - `unique` - whether to accept or not insertion of documents with duplicate keys,
  - `background`, `dropDups`, …

Indexes

- MongoDB provides different data-type indexes
  - Single field indexes
  - Compound field indexes
  - Multikey indexes (to index the content stored in **arrays**, MongoDB creates separate index entries for every element of the array)
  - Geospatial indexes (2d indexes with **planar** and 2dsphere with **spherical** geometry)
  - Text indexes (searching for **string** content in a collection, they do not store language-specific stop words, e.g., "the", "a", "or", and stem the words in a collection to only store root words
  - Hashed indexes (indexes the hash of the value of a field, they have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries)

Indexes

- Single field indexes
  - Support user-defined ascending/descending indexes on a single field of a document
- E.g.,
  - `db.orders.createIndex( {orderDate: 1} )`
- Compound field indexes
  - Support user-defined indexes on a set of fields
- E.g.,
  - `db.orders.createIndex( {orderDate: 1,  zipcode: -1} )`

Indexes

- MongoDB supports efficient queries of geospatial data
- Geospatial data are stored as:
  - GeoJSON objects: embedded document { <type>, <coordinate> }
    - E.g., location: `{type: "Point", coordinates: [-73.856, 40.848]}`
  - Legacy coordinate pairs: array or embedded document
    - `point: [-73.856, 40.848]`
- Fields with 2dsphere indexes must hold geometry data in the form of coordinate pairs or GeoJSON data.
  - If you attempt to insert a document with non-geometry data in a 2dsphere indexed field, or build a 2dsphere index on a collection where the indexed field has non-geometry data, the operation will fail.

Indexes

- Geospatial indexes
  - Two type of geospatial indexes are provided: `2d` and `2dsphere`
- A `2dsphere` index supports queries that calculate geometries on an earth-like sphere
- Use a `2d` index for data stored as points on a two-dimensional plane.
- E.g.,
  - `db.places.createIndex( {location: "2dsphere"} )`
- Geospatial query operators
  - $geoIntersects, $geoWithin, $near, $nearSphere

Indexes

- `$near` syntax:

```
{
   <location field>: {
     $near: {
       $geometry: {
          type: "Point" ,
          coordinates: [ <longitude> , <latitude> ]
       },
       $maxDistance: <distance in meters>,
       $minDistance: <distance in meters>
     }
   }
}
```

Indexes

- E.g.,
  - `db.places.createIndex( {location: “2dsphere”} )`
- Geospatial query operators
  - $geoIntersects, $geoWithin, $near, $nearSphere
- Geopatial aggregation stage
  - $near

Indexes

- E.g.,
  - ```
    db.places.find({location:
        {$near:
            {$geometry: {
              type: "Point",
              coordinates: [ -73.96, 40.78 ] },
             $maxDistance: 5000}
        }})
    ```
  - Find all the places within 5000 meters from the specified GeoJSON point, sorted in order from nearest to furthest
    ```

Indexes

- Text indexes
  - Support efficient searching for string content in a collection
  - Text indexes store only *root words* (no language-specific *stop words* or *stem*)
- E.g.,

```
db.reviews.createIndex( {comment: "text"} )
```

  - Wildcard ($**) allows MongoDB to index every field that contains string data
  - E.g.,

```
db.reviews.createIndex( {"$**": "text"} )
```

VIEWS

- 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**.
- Starting in version 4.2, MongoDB adds the $merge stage for the aggregation pipeline to create on-demand **materialized views**, where the content of the output collection can be updated each time the pipeline is run.
- **Read-only** views from existing collections or other views. E.g.:
  - excludes private or confidential data from a collection of employee data
  - adds computed fields from a collection of metrics
  - joins data from two different related collections

```
db.runCommand( {
  create: <view>, viewOn: <source>, pipeline: <pipeline>, collation: <collation> } )
```

- Restrictions
  - immutable Name
  - you can modify a view either by dropping and recreating the view or using the *collMod* comman