

7 MongoDB DML (Data Manipulation Language)

CS 425
Web Applications Development

Alex Petrovici

Content

1 Basic Queries

find() query

ICU Collation

Expression Operators

Regular Expressions

References

2 Updates

3 Aggregation Pipeline

4 Glossary

Basic Queries / find() query



MongoDB.

find() query

- The `find()` method accepts two parameters: *condition* and *projection*.
- Syntax: `var myCursor = db.ebooks.find(<condition> [, <projection>]);`
- Each parameter is a document (defined with `{...}`). Both are optional.
- An empty document (`{}`) is an unconditional selection (all documents are selected).
- The condition specifies restrictions for fields, separated by commas (conjunction `\wedge`).
- The basic `label:value` restriction is an equality test.
- More formally: returns documents containing all specified key-value pairs.
- **Example:**

```
db.ebooks.find({Name:"Alice"}); // returns all documents where Name is "Alice"  
db.ebooks.find({Name:"Alice"}, {Name:1, _id:0}); // returns the fields Name and _id
```

find() query

Basic Queries / find() query



mongoDB.

Namespace

- A BSON document is an object consisting of attributes (fields).
- Each attribute is accessed by its name (label).
- Sub-documents (name: { . . . }) use dot notation: car.licensePlate.
- Arrays also use dot notation, starting from position 0 to n-1.
- **Example:** Access Name of first element in Titles: Titles.0.Name
- To avoid syntax errors, write labels in quotes:

```
"Titles.0.Name" // returns the name of the first title
```

Basic Queries / find() query



Condition

- The projection is a sequence of inclusions (`field:1`) or exclusions (`field:0`), separated by commas and enclosed in curly braces.
- A projection is of a single type: it is either a specification of included fields or a specification of excluded fields, not both.
- The object identifier (`_id`) is a field that MongoDB adds to all documents. It does not need to be explicitly included (it is added by default to all queries), but it can be explicitly excluded (exceptionally, its exclusion (`_id:0`) can be specified in an inclusion projection).
- The projection cannot appear by itself (if you want to project an unconditional selection, the condition expression should be the empty document `{}`).
- **Example:**

```
db.ebooks.find( {}, {_id:0} ); // returns all documents except the _id field
```

find() query

Basic Queries / find() query



Examples:

- `db.ebooks.find({}, {name:1})` shows `_id` and name for all documents.
- `db.ebooks.find({}, {name:1, _id:0})` shows only name.
- `db.ebooks.find({LastName:"Dumas"}, {LastName:0, _id:0, Títulos:0})` excludes `_id`, `LastName`, and `Títulos`.
- When projecting arrays, limit displayed elements using `{$slice}`, `{$}`, `{$elemMatch}`:
 - array: `{$slice: X}` — first X elements (negative = last).
 - array: `{$slice: [J, X]}` — X elements starting at J.
 - `"array.$": 1` — first element matching the query condition.
 - array: `{$elemMatch: {$gt: 150}}` — element matching a new condition.

Basic Queries / find() query mongoDB.

Examples:

- `db.ebooks.find({}, {Titles: {$slice: -1}})`
Returns all documents, but only the last element of the `Titles` array.
- `db.ebooks.find({}, {Titles: {$slice: [0,5]}})`
Returns all documents, but only the first five elements of `Titles`.
- `db.ebooks.find({"Titles.Year":1844}, {"Titles.$":1})`
Returns documents where `Titles` contains a subdocument with `Year = 1844`. Projects `_id` and only the first matching element of `Titles`.
Note: `1 ≡ true`; `"$"` cannot be used with `0 (false)`.

Basic Queries / find() query



mongoDB.

- The conditional expression can be a restriction applied to an attribute, with the syntax `{attribute:restriction}`, or several restrictions separated by commas (the expression will be true when the logical AND of all restrictions holds).
- There are numerous conditional operators to define restrictions:
 - Equality `{$eq: value}`, inequality `{$ne: value}`
 - Greater than `{$gt: value}`, greater than or equal to `{$gte: value}`,
 - Less than `{$lt: value}`, less than or equal to `{$lte: value}`
 - Inclusion in a list `{$in: [v1, v2,\dots, vn]}`, exclusion `{$nin: [v1, v2,\dots, vn]}`
 - Existence of an element `{$exists: true}`; absence `{$exists: false}`
 - Type of element `{$type: BSONtype}`
(BSONtype is a number (1–255) or a label: double, string, object, array, objectId, bool, date, null, int, long, ...)
 - Size of array: `array_field:{$size: integer | restriction}`

Basic Queries / find() query

Example:

```
db.ebooks.find({Name: { $eq: "Alice" }, Pages: { $gt: 1500 }}, {Name:1, Pages:1, _id:0});
```

Basic Queries / `find()` query



mongoDB.

Logical Operators

There are four logical operators for building expressions:

- `{$and:[{exp1}, {exp2}, ..., {expn}]}` — Logical AND of all expressions
- `{$or:[{exp1}, {exp2}, ..., {expn}]}` — Logical OR of all expressions
- `{$nor:[{exp1}, {exp2}, ..., {expn}]}` — Logical NOR of all expressions
- `{$not: {exp}}` — Logical negation of the expression

If a restriction (with several conditions) is defined over an array of elements, the condition will be true if there is at least one element in the array that satisfies *each one* of the conditions individually. For example:

Here, a document will match if it contains a `Titles` array where at least one element has `Year` equal to 1848, and at least one (possibly different) element has `Name` equal to "L'Affaire Clemenceau".

```
db.ebooks.find({ "Titles.Year": 1848,  
    "Titles.Name": "L'Affaire Clemenceau" });
```

find() query

Basic Queries / find() query



mongoDB.

On the other hand, the **\$elemMatch** operator checks that there is a *single* element in the array that meets *all* the specified conditions:

```
db.ebooks.find({ Titles: { $elemMatch: { Year: { $gt: 1850, $lt: 1900 } } } });
```

This query returns the documents where the **Titles** array contains at least one element whose **Year** is strictly between 1850 and 1900.

find() query

Basic Queries / find() query



mongoDB.

- Multiple restrictions on the same field overwrite each other (last one applies):
 - `{Year: {$gt: 1900, $lt: 2000}}` — year between 1900 and 2000.
 - `{Year: {$gt: 1900}, Year: {$lt: 2000}}` — only checks year < 2000.
- To check if an array contains multiple values, use `$all`:
 - `{"Titles.Year": 1877, "Titles.Year": 1848}` — matches Year = 1848 only.
 - `{"Titles.Year": {$all: [1867, 1848]}}` — requires both years.
- Combine `$all` and `$elemMatch` for multiple matching profiles:

```
db.ebooks.find({ Titles: { $all: [
    { $elemMatch: { Year: { $gt: 1840, $lt: 1850 } } },
    { $elemMatch: { Year: { $gt: 1860, $lt: 1870 } } }
] } });
```

find() query

Basic Queries / find() query



mongoDB.

- If a restriction is applied to a subdocument, the condition is true only for documents that contain *exactly* that subdocument (with the same elements, values, and order).
- **Examples:** (based on documents from earlier slides in ebooks)
 - db.ebooks.find({Titles: {Name: "The Three Musketeers", Year: 1844}})
returns the first document.
 - db.ebooks.find({Titles: {Year: 1844, Name: "The Three Musketeers"}})
does *not* return any document! (order matters)
 - db.ebooks.find({Titles: {\$all: [{Name: "...", Year: 1844}, {...}]}})
returns the first document.
 - db.ebooks.find({Titles: {\$all: [{\$elemMatch: {Year: 1948}}, ...]}})
returns documents with a title in 1948 and another in 1967.
 - db.ebooks.find({"Titles.Year": 1844, "Titles.Name": "The Three Musketeers"})
returns the first document.



Basic Queries / ICU Collation

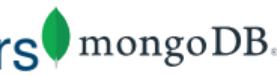


- MongoDB supports different ways of comparing string values (such as ignoring case, accents, etc.).
- The comparison specification `collation:{...}` can be added as an option when creating an object (collection, view, index), and as the `.collation({...})` method with operations like `find`, `delete`, `update`, or `aggregate`.
- The document includes the `locale` attribute, and additional options can also be added.

Attribute	Type	Description
locale	string	Language (e.g., Spanish "es"; binary comparison "simple").
strength	int	Comparison level (1 base, 2 with accents, 3 full; 4, 5).
caseLevel	boolean	Includes case comparison at levels 1 and 2.
numericOrdering	boolean	Compare numeric chars as numbers or strings (default).
alternate	string	Spaces and punctuation (<i>non-ignorable/shifted</i>).
maxVariable	string	<i>alternate</i> ignores only spaces or punctuation.



Basic Queries / Expression Operators



- `$mod: [X, Y]` — Remainder of integer division (returns true if *attribute MOD X = Y*).
- `$regex: ...` — Regular expressions on string attributes.
- `$expr: ...` — Allows use of aggregation expressions (such as in `$match`); enables more powerful queries.
- `$jsonSchema: ...` — Enforces document format (e.g., required fields, type constraints).

Example:

```
{ $jsonSchema: { required: ["author", "titles"], properties: { author: {  
    bsonType: "string" }}}}
```

https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#op._S_jsonSchema

- `$text: ...` — Performs text search (requires a text index).
- `$where: ...` — General-purpose evaluation operator (executes JavaScript code).



Basic Queries / Regular Expressions



- { field: { `\$regex: 'pattern'`, `\$options: 'imxs'` } }
- { field: { `\$regex: /pattern/`, `\$options: 'imxs'` } }
- { field: { `\$regex: /pattern/im` } }
- { field: `/pattern/im` } (*the only valid notation for use with \$in*)

i: Case-insensitive matching.

m: Multiline mode (anchors ^ and \$ match at the start/end of each line, separated by \n).

x: Ignores whitespace and # comments included in the pattern.

s: The dot metacharacter matches all characters, including newlines (\n).

```
db.ebooks.find({ Author: { $regex: /^Ale/ } })
```

Basic Queries / Regular Expressions



mongoDB.

- The generic evaluation operator is `$where{JavaScript code}`.
- In the JavaScript code, each document is accessed via `this` or `obj`.
- Although `$where` is flexible, it does **not** operate on subdocuments and is less efficient than native query operators.
- **Examples:**

```
db.ebooks.find({$where:"this.Author.slice(0,3)=='Ale'"})
```

- If `$where` is the only query condition, the operator can be omitted:

```
db.ebooks.find({"this.Author.slice(0,3) == 'Ale'"})
```

- It can also take a function as a parameter (which must return a boolean):

```
db.ebooks.find({$where: function() { return  
    → (obj.Author.slice(0,3)=='Ale') }})
```

Basic Queries / Regular Expressions



mongoDB.

- Native MongoDB operators allow using indexes and other efficient retrieval strategies (filtering blocks to read). The `$where` clause does *not*, as it implies running interpreted code on each document—thus it is inefficient.
- **Lazy evaluation:** if there are multiple conditions in a query, `$where` is evaluated last.
- For efficiency, avoid using `$where` whenever possible. Only use it if no other operator or combination can express the query.
- For more details on the `$where` operator, see:
<http://docs.mongodb.org/manual/reference/operator/query/where/>

Basic Queries / References



- If you are familiar with relational databases, you may notice the lack of relationships (links) between collections and queries spanning multiple collections.
- For example, if we have a collection for people and another for cars, it is natural for each car to include a reference to its owner.
- DML usually includes mechanisms to resolve such references (e.g., SQL JOIN).
- However, MongoDB is based on an “unstructured” paradigm, and data is generally expected to be denormalized in a single monolithic collection.
- As a result, operational manipulation may be less efficient, but analytical queries can be more efficient.
- Nevertheless, MongoDB has evolved to incorporate other paradigms (e.g., Atlas provides ACID features).

Basic Queries / References



- Although it is not inherent to its design, MongoDB does support references (as document identifiers in another collection):
`DBRef: {"$ref": ..., "$id": ..., "$db": ...}`
- If cross-collection queries are needed, these are typically resolved in application code (host language). First, a query retrieves an array of references, and then a second query uses that result to combine data (for example, using the `$in` operator).
- For example, following the previous scenario, we first obtain the IDs (such as DNI) of car owners, store them in an array variable, and then execute another query that selects the documents from the people collection whose DNI attribute is contained in that array.
- Since this is a manual process, it can be done with any identifier (MongoDB usually works with `_id`, but without integrity constraints!).
- Since MongoDB 3.2, a *left outer join* is available as an aggregation pipeline operator!
- And since version 4, MongoDB (Atlas) provides ACID transactions!

Updates / mongoDB

- In addition to inserting documents, there are methods for deleting and updating documents and collections.
- To delete an entire collection (including all its documents):
 - `db.col.drop()`: deletes the collection `col` and all its documents.
- DML for deleting individual documents:
 - `db.col.remove(query)`: removes all documents that match the given query.
Example: `db.ebooks.remove({LastName: "Dumas"})`
- If `query` is `{}`, all documents in the collection will be deleted.
- Note: Emptying a collection is not the same as dropping it; in many cases, it is more efficient to drop (and then regenerate indexes, etc.).
- `remove` accepts a second parameter `{justOne:1}` to delete only a single matching document.

Updates / mongoDB

- `db.col.save(doc)`: saves the specified document;
 - If the document includes the identifier field `_id`, performs an update;
 - Otherwise, inserts the document as new.
- `db.col.update({query}, {updating} [, {options}])`:
 - Updates the documents matching the query with the changes described in `updating`.
 - The `updating` clause can be a complete document (which will replace the matched document) or a set of changes to apply.
 - By default, only the first found document is updated.
 - Options:
 - `multi:1`: updates all matching documents.
 - `upsert:1`: if no document matches the query, inserts a new document (fields are taken from both `updating` and `query`).

Updates / mongoDB.

- To describe a sequence of changes, the following operators are available:
- `$set:{...}` accepts pairs of field and value. If the field exists, its value is updated; if it does not exist, it is created with the specified value. To reference a subfield, use `subdocument.field`, and to reference the n -th position of an array, use `array.n` (e.g., `array.8`).
- `$setOnInsert:{...}` works similarly to `$set`, but is ignored on updates; it only has an effect when inserting documents during upsert operations.
- `$unset:{'old':'""...}'` removes the specified fields.
- `$rename:{'old':'new',...}` renames fields.
- `$inc:{field:x,...}` increments the specified field by x (can be positive or negative). There is also `$mul` which multiplies instead of adding.
- `$max:{...}` only updates the field if the specified value is greater. There is also `$min`, which performs the opposite.
- `$currentDate:{field:1}` updates the field with the current date.

Updates / mongoDB.

- Update operators for arrays:
 - `$pop:{...}` removes the first (or the last) element of the array.
 - `$pull:{field:query}` removes elements from the array that match the query.
 - `$pullAll:{field:[list]}` removes all elements included in the given list from the array.
 - `$push:{...}` inserts the specified elements into the array.
 - `$addToSet:{...}` inserts the specified elements only if they do not already exist in the array.
- When adding elements to arrays, the following modifiers can be used:
 - `$position:x` inserts elements at a specific position (x) in the array.
 - `$each:[list]` inserts all elements from the specified list into the array.
 - `$slice:n` limits (truncates) the array to a maximum size of *n* elements.
 - `$sort:{...}` if the array contains values, sorts in ascending (1) or descending (-1) order; if the array contains documents, sorts them by a specified field, e.g., `{field:1}`.
- For a comprehensive reference on update operators, see:
<http://docs.mongodb.org/manual/reference/operator/update/>

Updates / mongoDB

- Remember that MongoDB is designed for bulk queries, not transactional processing; updates should generally be used for transformation and data cleaning.
- Additionally, if you need to perform something “special”, you can always use JavaScript.
- For example, let’s see how to update a document by reusing attributes:

```
// This example restructures the documents: creates the 'Author' attribute from existing
→ fields
db.ebooks.find().snapshot().forEach(function(doc) {
    doc.Author = { Name: doc.Name, Surname: doc.Surname };
    // Remove the original attributes
    delete doc.Name;
    delete doc.Surname;
    // Save the modified document
    db.ebooks.save(doc);
});
```

Updates / mongoDB®

- Notice that in this case, it is simply a renaming operation, which can be done directly with:

```
db.ebooks.update(  
  {},  
  { $rename: { 'Name': 'Author.Name', 'Surname': 'Author.Surname' } },  
  false,  
  true  
)
```

Aggregation / Pipeline



- An aggregated (or grouped) query summarizes data from a collection.
- In MongoDB, there are three main types of aggregation: simple, pipeline, and map-reduce.
- Simple aggregations include the cardinality methods, distinct documents (`distinct`), and the generic grouping (`group`).
- The method `db.collection.count([query])` returns the number of documents that satisfy the query (or the total number of documents in the collection if the query is omitted). Using `db.collection.find([query]).count()` yields the same result, but through a different mechanism.
- The method `db.collection.distinct(field, [query])` returns the distinct values for the specified field in the documents of the collection (optionally restricted to documents matching the query).

Aggregation / Pipeline



- **How does aggregation (the aggregation pipeline) work?**
- This method takes as a parameter either a single stage or a list of stages.
- The first stage takes the collection and transforms it, producing a new collection.
- Each subsequent stage receives the output of the previous stage and produces another transformed collection.
- After all stages are complete, a cursor containing the final result collection is returned.
- The aggregation pipeline works in several stages:
 - ① Starts by creating `res(0)`, which is the original collection.
 - ② For each stage, checks if any stages are left; if yes, applies stage n to the result of the previous stage (`res(n-1)`), producing `res(n)`.
 - ③ This process repeats until there are no stages left.
 - ④ Finally, the last result collection `res(n)` is returned.
- **Syntax:** `db.collection.aggregate([stage1, stage2, ...])`
- This method supports large and sharded collections.

Aggregation / Pipeline



Common Aggregation Pipeline Stage Operators

`$match:{query}`

Filters the collection by applying the given selection.

Aggregation / Pipeline



Common Aggregation Pipeline Stage Operators

\$project:{spec}

Projects existing fields (just like in the `find` method), reassigns, and/or creates new fields (accessing values with “`$attribute`”).

Supports the operator `$cond:{if:expr, then:value, else:value}`.

Example:

```
$project: {  
    "ID": 1,  
    "version": {$literal: 1},  
    "Name": {$concat:[ "$Name", " $Surname" ]},  
    "inDate": {  
        $cond: {  
            if: {$isArray: "$inDate"},  
            then: {$arrayElemAt: ["$inDate", 0]},  
            else: "$inDate"  
        }  
    }  
}
```

Aggregation / Pipeline



Common Aggregation Pipeline Stage Operators

\$unwind: array

Unwinds an array field, creating a document for each array element (element: position, plus the rest of the document).

By default, omits documents with empty arrays/missing or other datatypes.

Example:

```
$unwind: {  
  path: "$Names",  
  includeArrayIndex: "which",  
  preserveNullAndEmptyArrays: true  
}
```

Aggregation / Pipeline



Common Aggregation Pipeline Stage Operators

`$group:{spec}`

Creates a **grouped collection** (groups the collection by criteria). spec must include the grouping key (`_id`) and aggregated fields.

`$out: "col_name"`

Creates and stores a new collection with the received documents.

Aggregation / Pipeline



- **Expression Operators:** Used to add fields to the resulting document. Syntax:
`$operator: [arg1, arg2, ..., argn]`
- Arithmetic: addition (`$add`), subtraction (`$subtract`), multiplication (`$multiply`), division (`$divide`), modulus (`$mod`)
- **String Operators:** concatenation (`$concat`), substring (`$substr`), lowercase (`$toLower`), uppercase (`$toUpper`), comparison (`$strcasecmp`)
- **Comparison and Logical Operators:** `$and`, `$or`, `$not`, `$eq`, `$ne`, `$gt`, `$lt`, etc.
- **Set Operators:** equality (`$setEquals`), inclusion (`$setIsSubset`), union (`$setUnion`), intersection (`$setIntersection`), difference (`$setDifference`)

Aggregation / Pipeline



- Aggregation Operators for `$group`:
 - `$sum`: sum of the group's values (one value per document)
 - `$avg`: average of the group's values (one value per document)
 - `$max`: maximum value in the group
 - `$min`: minimum value in the group
 - `$first`: value of the first document in the group
 - `$last`: value of the last document in the group
 - `$push`: array containing all values from the group (one value per document)
 - `$addToSet`: similar to `$push`, but discards duplicate values

- For a complete list, see:

<https://docs.mongodb.com/manual/reference/operator/aggregation/>

Basic Queries
oooooooooooo
o
o
oo
oo

Updates
oooooo

Aggregation
oooooooo

Glossary

Glossary

Abbreviations

Terms

Glossaries

Basic Queries
oooooooooooo

Updates
oooooo

Aggregation
oooooooo

Glossary

Glossary

Abbreviations

Terms

Glossary / Terms I

Basic Queries
oooooooooooo

Updates
oooooo

Aggregation
oooooooo

Glossary

Glossary

Abbreviations

Abbreviations

Glossary / Abbreviations I