MongoDB Operators Cheat Sheet

A comprehensive guide to MongoDB Comparison, Logical, Element, Array, Evaluation, and Update operators.

1. Comparison Operators

These are used to **compare field values**. You mostly use them inside `find`, `findOne`, `update`, `delete` filters.

1.1 `$eq` – Equal To

Explanation

- `$eq` means **"equal to this value"**.
- Matches documents where a field's value **exactly equals** the given value (type matters— `"10"` is not equal to `10`).

Implementation

```
// Syntax pattern
{ fieldName: { $eq: value } }

// Often you can also just write:
{ fieldName: value } // MongoDB treats this like $eq
```

Practical Example (Users collection) Find all users whose role is exactly "student":

```
db.users.find({ role: { $eq: "student" } });

// Or simply:
db.users.find({ role: "student" });
```

Real-life scenario: You're building a dashboard for only "admin" users. In your backend route, you query:

```
const admins = await db.collection("users").find({ role: { $eq: "admin" } }).toArray();
```

1.2 `$ne` – Not Equal To

Explanation

- `$ne` means **"not equal to this value"**.
- It matches documents where the field's value is anything except the given value (including null or missing fields, unless you combine with other conditions).

Implementation

```
{ fieldName: { $ne: value } }
```

Practical Example Get all users who are not students:

```
db.users.find({ role: { $ne: "student" } });
```

Real-life scenario: You want a list of all non-admin users (for sending a general announcement).

```
db.users.find({ role: { $ne: "admin" } });
```

1.3 `$gt` – Greater Than

Explanation

- `$gt` means **"greater than this value"**.
- Works with numbers, dates, and strings (lexicographically).

Implementation

```
{ fieldName: { $gt: value } }
```

Practical Example (Orders collection) Find all orders with amount greater than 1000:

```
db.orders.find({ amount: { $gt: 1000 } });
```

Real-life scenario: To see high-value orders (for VIP customers, fraud checks, etc.).

1.4 `$gte` – Greater Than or Equal To

Explanation

- `$gte` = greater than or equal to.
- Includes the boundary value.

Implementation

```
{ fieldName: { $gte: value } }
```

Practical Example Find users who are 18 years or older:

```
db.users.find({ age: { $gte: 18 } });
```

Real-life scenario: You must only show content or features to 18+ users (legal / compliance).

1.5 `$lt` – Less Than

Explanation

- `$lt` means **"less than this value"**.

Implementation

```
{ fieldName: { $lt: value } }
```

Practical Example Find orders with amount less than 500:

```
db.orders.find({ amount: { $lt: 500 } });
```

Real-life scenario: Filter small-ticket orders to design micro-offers or discounts specifically for them.

1.6 `$lte` – Less Than or Equal To

Explanation

- `$lte` = less than or equal to.

Implementation

```
{ fieldName: { $lte: value } }
```

Practical Example Find users younger than or equal to 21:

```
db.users.find({ age: { $lte: 21 } });
```

Real-life scenario: Offering student discounts or specific benefits to users up to a certain age limit.

1.7 `$in` – Value is in a List

Explanation

- `$in` checks if a field's value matches any value from an array of values.
- Instead of writing multiple `$or` conditions, you give a list.

Implementation

```
{ fieldName: { $in: [value1, value2, ...] } }
```

Practical Example Find users in Pune or Mumbai or Delhi:

```
db.users.find({
  city: { $in: ["Pune", "Mumbai", "Delhi"] }
});
```

Real-life scenario: You're sending notifications only to specific cities where your service is live.

1.8 `$nin` – Value is NOT in a List

Explanation

- `$nin` is the opposite of `$in`.
- Matches documents where the field's value is not any of the values in the array.

Implementation

```
{ fieldName: { $nin: [value1, value2, ...] } }
```

Practical Example Get users not living in metro cities:

```
db.users.find({
  city: { $nin: ["Mumbai", "Delhi", "Bengaluru", "Chennai"] }
});
```

Real-life scenario: You want to offer a special bonus to users in smaller towns/rural areas only.

2. Logical Operators

Logical operators combine multiple conditions.

2.1 `$and` – All Conditions Must Be True

Explanation

- `$and` combines conditions where all conditions must be satisfied.
- Often you can omit `$and` because MongoDB treats multiple key-value pairs at top level as AND.

Implementation

```
{
  $and: [
    { age: { $gte: 18 } },
    { city: "Pune" }
  ]
}

// Equivalent shorthand:
{ age: { $gte: 18 }, city: "Pune" }
```

Practical Example Find users (Age ≥ 18 AND City = Pune):

```
db.users.find({
  $and: [
    { age: { $gte: 18 } },
    { city: "Pune" }
  ]
});
```

Real-life scenario: You want to find eligible users in a specific city for a local offline event.

2.2 `$or` – At Least One Condition Must Be True

Explanation

- `$or` means "any one of these conditions can be true".
- Used when you want to match multiple alternative filters.

Implementation

```
{
  $or: [
    { city: "Pune" },
    { city: "Mumbai" }
  ]
}
```

Practical Example Find users either from Pune or with age < 20:

```
db.users.find({
  $or: [
    { city: "Pune" },
    { age: { $lt: 20 } }
  ]
});
```

Real-life scenario: Show an offer to Pune residents or young students across India.

2.3 `$not` – Negate a Condition

Explanation

- `$not` inverts the result of another condition.
- Usually used inside another operator like `$regex`, `$gt`, etc.

Implementation

```
{ fieldName: { $not: { <another condition> } } }
```

Practical Example Find all users whose age is NOT greater than 18 (so ≤ 18):

```
db.users.find({
  age: { $not: { $gt: 18 } }
});
// Equivalent logically: { age: { $lte: 18 } }
```

Real-life scenario: You want to explicitly exclude records matching a complex condition, like “not matching a certain regex pattern”.

2.4 `$nor` – None of the Conditions Should Be True

Explanation

- `$nor` = NOT OR.
- Matches documents where all conditions are false.

Implementation

```
{
  $nor: [
    { city: "Pune" },
    { age: { $lt: 18 } }
  ]
}
```

Practical Example Find users not in Pune and not younger than 18:

```
db.users.find({
  $nor: [
    { city: "Pune" },
    { age: { $lt: 18 } }
  ]
});
```

Real-life scenario: You want to avoid a particular city and also avoid minors at the same time.

3. Element Operators

These work on presence of fields and data types.

3.1 `$exists` – Field Exists or Not

Explanation

- `$exists: true` – field must exist (can be null).
- `$exists: false` – field must not exist.

Implementation

```
{ fieldName: { $exists: true } }
{ fieldName: { $exists: false } }
```

Practical Example Find users who have set their phone number:

```
db.users.find({ phone: { $exists: true } });
```

Find users who haven't provided a phone number:

```
db.users.find({ phone: { $exists: false } });
```

Real-life scenario: You might want to force phone number verification only for users who don't have phone numbers set.

3.2 `$type` – Match by Data Type

Explanation

- `$type` matches documents where the field has a specific BSON type (string, number, array, etc.).

Implementation

```
{ fieldName: { $type: "string" } }
// or numeric codes like 2 for string, 16 for int, etc.
```

Practical Example Find users where age is stored as a string (bad schema, needs fixing):

```
db.users.find({ age: { $type: "string" } });
```

Real-life scenario: During data cleanup/migration, you want to find incorrectly typed data.

4. Array Operators

Used when your field value is an array, like:

```
{
  name: "Prathamesh",
  skills: ["MongoDB", "Node.js", "React"]
}
```

4.1 `$all` – Array Contains All These Values

Explanation

- `$all` checks whether the array field contains all elements in the provided list.

Implementation

```
{ arrayField: { $all: [value1, value2, ...] } }
```

Practical Example Find users who know both MongoDB and Node.js:

```
db.users.find({
  skills: { $all: ["MongoDB", "Node.js"] }
});
```

Real-life scenario: You want to find candidates that match all required skills for a project.

4.2 `$size` – Array Length

Explanation

- `$size` matches arrays whose length is exactly N.

Implementation

```
{ arrayField: { $size: number } }
```

Practical Example Find users having exactly 3 skills:

```
db.users.find({
  skills: { $size: 3 }
});
```

Real-life scenario: Maybe you want to show a message like “Add more skills to improve your profile” to users with too few skills.

4.3 `$elemMatch` – Match an Element Inside Array with Conditions

Explanation

- `$elemMatch` is used when array elements are objects, and you want conditions on those inner fields.

Implementation

```
{ arrayField: { $elemMatch: { condition1, condition2, ... } } }
```

Practical Example Find students who have any exam with subject = "Maths" and score ≥ 85:

```
db.students.find({
  exams: {
    $elemMatch: {
      subject: "Maths",
      score: { $gte: 85 }
    }
  }
```

```
});
```

Real-life scenario: Finding students who meet a particular subject-specific performance criteria.

5. Evaluation Operator – `$regex`

5.1 `$regex` – Pattern Matching

Explanation

- `$regex` lets you match strings using regular expressions.
- Similar to LIKE in SQL but more powerful.

Implementation

```
{ fieldName: { $regex: /pattern/, $options: "i" } } // i = case-insensitive
```

Practical Example Find users whose name starts with "Pra" (case insensitive):

```
db.users.find({
  name: { $regex: /^pra/i }
});
```

Real-life scenario: Implementing search-as-you-type functionality in your app – user types “pra”, you search names starting with “pra”.

6. Update Operators

These are used in `updateOne`, `updateMany`, `findOneAndUpdate`, etc.

6.1 `$set` – Set / Overwrite a Field

Explanation

- `$set` updates the value of a field.
- If the field does not exist, it creates it.
- If it exists, it overwrites the previous value.

Implementation

```
db.collection.updateOne(
  { filter },
  { $set: { field1: value1, field2: value2 } }
);
```

Practical Example Set a user’s city to "Pune":

```
db.users.updateOne(
  { name: "Prathamesh" },
  { $set: { city: "Pune" } }
);
```

Real-life scenario: User updates their profile info (city, bio, etc.) in your app.

6.2 `$unset` – Remove a Field

Explanation

- `$unset` removes a field from the document.
- You don't care about the exact value; you just want it gone.

Implementation

```
db.collection.updateOne(
  { filter },
  { $unset: { fieldName: "" } } // value doesn't matter
);
```

Practical Example Remove a tempToken field after verification:

```
db.users.updateOne(
  { email: "user@example.com" },
  { $unset: { tempToken: "" } }
);
```

Real-life scenario: Cleaning up one-time fields like OTP, password reset token after they're used.

6.3 `$inc` – Increment (or Decrement) a Number

Explanation

- `$inc` adds or subtracts a numeric value.
- If the field doesn't exist, MongoDB will create it starting from 0 then apply the increment.

Implementation

```
db.collection.updateOne(
  { filter },
  { $inc: { numericField: amount } }
);
```

Practical Example Increase a user's loginCount by 1:

```
db.users.updateOne(
  { _id: userId },
  { $inc: { loginCount: 1 } }
```

```
);
```

Decrease product stock by 2:

```
db.products.updateOne(
  { _id: productId },
  { $inc: { stock: -2 } }
);
```

Real-life scenario:

- Tracking how many times a user logged in.
- Updating quantity in a shopping cart or inventory.

6.4 `$push` – Add Item to Array

Explanation

- `$push` appends a value to an array field.
- If the field doesn't exist, MongoDB creates it as an array.

Implementation

```
db.collection.updateOne(
  { filter },
  { $push: { arrayField: value } }
);
```

Practical Example Add a new skill to user:

```
db.users.updateOne(
  { name: "Prathamesh" },
  { $push: { skills: "Node.js" } }
);
```

Real-life scenario: User adds a new skill, tag, or interest in their profile.

6.5 `$addToSet` – Add Unique Item to Array

Explanation

- `$addToSet` adds a value to an array only if it is not already present.
- Prevents duplicates.

Implementation

```
db.collection.updateOne(
  { filter },
  { $addToSet: { arrayField: value } }
```

```
);
```

Practical Example Add "MongoDB" to a user’s skills only if it's not already there:

```
db.users.updateOne(
  { name: "Prathamesh" },
  { $addToSet: { skills: "MongoDB" } }
);
```

Real-life scenario: Managing tags or favorites where duplicates make no sense.

6.6 `$pull` – Remove Matching Items from Array

Explanation

- `$pull` removes all array elements that match a given condition.

Implementation

```
db.collection.updateOne(
  { filter },
  { $pull: { arrayField: valueOrCondition } }
);
```

Practical Example Remove "MongoDB" from user’s skills:

```
db.users.updateOne(
  { name: "Prathamesh" },
  { $pull: { skills: "MongoDB" } }
);
```

Or remove all exam entries where score < 40:

```
db.students.updateOne(
  { name: "Prathamesh" },
  {
    $pull: {
      exams: { score: { $lt: 40 } }
    }
  }
);
```

Real-life scenario:

- Removing a tag user unselected.
- Removing failed attempts or outdated tokens.

7. How to Think About Operators While Coding

1. **Comparison + Logical operators** together define filters: “Which documents do I want?”
2. **Element + Array operators** help you work with data shape and nested structures.
3. **Update operators** modify these matched documents in a safe, atomic way.