Queries — Filter, Sort, Operators

Ask sharper questions: comparison operators ($gte, $in, $ne), sorting, and LIMIT/SKIP paging over your documents.

Introduction

`find({ city: 'Sahiwal' })` was a good first question — but real questions have edges: *marks of 90 or more*, *students NOT from Karachi*, *the top two, in order*, *page two of the list*. MongoDB answers all of them with **query operators** — small words starting with `$` that turn a filter from an exact match into a real question. This lesson is the MongoDB twin of SQL's DB.4 (WHERE, ORDER BY, LIMIT): comparison operators, list operators, sorting, and the limit/skip pair that builds pages.

(Roman Urdu: $gte matlab "is se zyada ya barabar" — operator wo chhota lafz hai jo sawal ko tez karta hai.)

Key points

- **Comparison operators do the asking:** `$gt` (greater), `$gte` (greater or equal), `$lt`, `$lte`, `$ne` (not equal), `$eq`. They sit *inside* the field's value: `{ marks: { $gte: 90 } }` reads "marks where value ≥ 90".
- **Two conditions on one field combine:** `{ marks: { $gt: 70, $lt: 90 } }` — both must hold (an AND inside one pair of braces). Conditions on *different* fields are also AND by default: `{ city: 'Sahiwal', marks: { $gte: 90 } }`.
- **$in and $nin ask about lists:** `{ city: { $in: ['Sahiwal', 'Karachi'] } }` matches either value — SQL's `IN` in document clothes. `$nin` is its opposite. `$exists` asks whether a field is present at all (`{ scholarship: { $exists: true } }` — remember Hira), and `$regex` searches text by pattern.
- **sort() arranges the answer:** `.sort({ marks: -1 })` is biggest first (descending), `.sort({ marks: 1 })` is smallest first. The number is the direction — 1 up, −1 down — the same trick SQL's ASC/DESC does with words.
- **limit() and skip() build pages:** `.limit(2)` takes the first two; `.skip(2).limit(2)` takes the *next* two — page two. Sort first, always: pages of an unordered list are meaningless, because the order could differ between two reads.
- **countDocuments(filter) counts without fetching:** the same filter, a number instead of documents — for totals, and for the safety habit from MG.3.

Code examples

Example 1 — The 90 club: $gte at work

```
db.students.find({ marks: { $gte: 90 } })
```

What it does: One operator turns "show me the register" into "show me the best". Only documents whose `marks` field is 90 or more survive the filter.

What you will see:

```
[
  {
    _id: ObjectId('650000000000000000000002'),
    name: 'Sara Khan',
    class: '10-A',
    marks: 92,
    city: 'Lahore'
  },
  {
    _id: ObjectId('650000000000000000000004'),
    name: 'Hira Shah',
    class: '10-B',
    marks: 95,
    city: 'Sahiwal',
    scholarship: true
  }
]
```

Sara (92) and Hira (95) — the same two names SQL's `WHERE marks >= 90` returned in DB.1. Different language, same truth. Notice the answers arrive in *insert order*, not marks order — a filter never promises arrangement. That is the next example's job.

Example 2 — A list question: $in with two cities

```
db.students.find({ city: { $in: ['Sahiwal', 'Karachi'] } })
```

What it does: Instead of running two queries and stitching the answers by hand, `$in` hands the field a list of acceptable values. Either city matches; everything else is filtered out.

What you will see:

```
[
  {
    _id: ObjectId('650000000000000000000001'),
    name: 'Ali Raza',
    class: '10-A',
    marks: 88,
    city: 'Sahiwal'
  },
  {
    _id: ObjectId('650000000000000000000004'),
    name: 'Hira Shah',
    class: '10-B',
    marks: 95,
    city: 'Sahiwal',
    scholarship: true
  },
  {
    _id: ObjectId('650000000000000000000005'),
    name: 'Usman Tariq',
    class: '9-A',
    marks: 64,
    city: 'Karachi'
  }
]
```

Ali and Hira from Sahiwal, Usman from Karachi — three documents. Try it with your eyes first on any new filter: *predict the names, then run.* A query you can predict is a query you understand; that habit catches more errors than any tool.

Example 3 — Order and pages: sort, limit, skip

```
db.students.find().sort({ marks: -1 }).limit(2)
db.students.find().sort({ marks: 1 }).skip(2).limit(2)
```

What it does: The first command builds the **topper list**: sort by marks descending, keep two. The second builds **page two** of the merit list in rising order: sorted ascending (64, 76, 88, 92, 95), skip the first two (Usman 64, Bilal 76), take the next two.

What you will see:

```
[
  {
    _id: ObjectId('650000000000000000000004'),
    name: 'Hira Shah',
    class: '10-B',
    marks: 95,
    city: 'Sahiwal',
    scholarship: true
  },
  {
    _id: ObjectId('650000000000000000000002'),
    name: 'Sara Khan',
    class: '10-A',
    marks: 92,
    city: 'Lahore'
  }
]
[
  {
    _id: ObjectId('650000000000000000000001'),
    name: 'Ali Raza',
    class: '10-A',
    marks: 88,
    city: 'Sahiwal'
  },
  {
    _id: ObjectId('650000000000000000000002'),
    name: 'Sara Khan',
    class: '10-A',
    marks: 92,
    city: 'Lahore'
  }
]
```

Hira and Sara on the podium; Ali and Sara on page two. Every app list you have ever scrolled — products, videos, messages — is sort + skip + limit underneath, exactly like this.

Quick recap

- Operators ask the question: `$gt/$gte/$lt/$lte/$ne` compare, `$in/$nin` check lists, `$exists` checks presence, `$regex` searches text.
- Multiple conditions are AND by default — across fields and inside one field's braces.
- `.sort({ field: 1 })` rises, `.sort({ field: -1 })` falls; **sort before you page**.

- `.skip(n).limit(m)` is a page; `countDocuments(filter)` counts the same question without fetching it.

Common mistakes

- **Writing the operator without the $:** `{ marks: { gte: 90 } }` does not error — it searches for documents whose marks field is an object `{ gte: 90 }`, finds none, and returns an empty list in silence. An empty answer is not always "no data"; sometimes it is a missing dollar sign.
- **Expecting the filter to sort:** `find` answers in natural (roughly insert) order. Beginners print "top students" that are just the *first* students. No `.sort()`, no order — write it on your hand.
  (Roman Urdu: Sort likhna bhool jao to list apni marzi ki tartib mein aati hai — filter tartib nahi deta.)
- **Paging without sorting:** skip/limit over an unordered collection can repeat or lose documents between pages as data changes. Sort by something stable (marks, then name) before paging a real app.
- **$ne surprises with missing fields:** `{ scholarship: { $ne: true } }` also matches documents that have *no* scholarship field at all. In a flexible-shape database, "not true" and "not present" are different answers — `$exists` is how you separate them.

Practice

1. Write the filter for students with marks between 70 and 90 (both ends included), and predict the names before running it. Then sort the same answer by name, A to Z.
2. Build page one and page three of the class list: sort by marks descending, two students per page. Which names land on page three, and why is there only one?
3. Hira is the only student with a `scholarship` field. Write two different queries that find exactly her — one with `$exists`, one without it — and say which one breaks if another student later wins a scholarship too.

Practice quiz

1. How do you ask for marks of 90 or more in a MongoDB filter?

- { marks: 90 }
- { marks: { $gte: 90 } }, with the operator inside the field's value
- { marks: { gte: 90 } }, without the dollar sign
- { $gte: { marks: 90 } }

Answer: { marks: { $gte: 90 } }, with the operator inside the field's value

2. In .sort({ marks: -1 }), what does -1 mean?

- Smallest first
- Biggest first, descending; 1 rises, -1 falls
- Sort by the last field
- Remove the marks field

Answer: Biggest first, descending; 1 rises, -1 falls

3. How do you build page two of two-per-page results, and what must come first?

- .limit(2) alone
- .skip(2).limit(2), and sort first always, because pages of an unordered list are meaningless
- .skip(2) alone
- Pages are built with find only

Answer: .skip(2).limit(2), and sort first always, because pages of an unordered list are meaningless

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/mongodb-queries · More free tools and notes at https://scopetreasure.com