How to Easily Query MongoDB Using Greater Than and Less Than Operators

Authored by

stats writer

November 30, 2025

Filtering data is a fundamental requirement of any modern database system. In the world of MongoDB queries, this filtering capability is powerfully implemented through a set of expressive comparison operators. These operators allow developers and analysts to precisely select documents based on specific field values relative to a given threshold, making operations like range queries straightforward and efficient.

The core of relational comparison in MongoDB revolves around the greater than (>) and less than (<) concepts, which are mapped to specific dollar-prefixed operators within the query document. Understanding how to correctly implement these operators--specifically **$gt** (greater than) and **$lt** (less than)--is critical for tasks ranging from analyzing sales figures and managing time-series data to filtering user profiles based on age or score. Unlike traditional SQL, MongoDB utilizes a declarative JSON-like syntax (or BSON internally) where the condition is passed as a nested object, providing remarkable flexibility.

This guide will serve as an expert resource for harnessing the full potential of these fundamental comparison tools. We will explore not only the basic syntax for queries involving single comparisons (e.g., finding all items greater than 20) but also complex scenarios requiring combined conditions (AND logic) and alternative conditions (OR logic). Furthermore, we will demonstrate how these operators handle various data types, including numbers, dates, and even strings, ensuring you can build robust and efficient data retrieval strategies.

Understanding MongoDB Comparison Operators

MongoDB provides four key comparison operators that correspond directly to standard mathematical inequalities. These operators are essential components of the query document used within the `db.collection.find()` method. They allow precise control over data selection, determining whether a field's value should be strictly greater than, strictly less than, or inclusive of the target value.

Using the correct operator is vital for accurate data retrieval, especially when dealing with boundaries in numerical ranges or date ranges. Below is the complete list of available comparison operators and their definitions:

$lt: Less than. Selects documents where the value of the field is strictly less than (<) the specified value.
$lte: Less than or equal. Selects documents where the value of the field is less than or equal to (≤) the specified value.
$gt: Greater than. Selects documents where the value of the field is strictly greater than (>) the specified value.
$gte: Greater than or equal. Selects documents where the value of the field is greater than or

equal to (≥) the specified value.

These operators are utilized by constructing a query object where the field name is the key, and the value is a nested object containing the operator and the comparison value. For example, to query a field named `score` to be greater than 100, the syntax would be `{ score: { $gt: 100 } }`. This structure ensures that MongoDB can efficiently parse and execute the necessary filters against the underlying data.

We will now explore four primary methodologies for applying these operators, moving from simple single-condition queries to complex combined queries involving both **$gt** and **$lt** conditions, demonstrating the flexibility of the MongoDB query language.

Implementing Single-Condition Range Queries

The simplest application of comparison operators involves filtering documents based on a single condition for a specific field. This is the foundation for almost all range-based data retrieval in MongoDB. We always use the `db.collection.find()` method, passing the query criteria object as the first argument.

Method 1: Greater Than Query (Using $gt)

To retrieve all documents where a designated field, such as `field1`, holds a value strictly larger than a specific number (e.g., 25), we utilize the **$gt** operator. This structure is highly efficient for setting a lower bound on your data selection.

db.myCollection.find({field1: {$gt:25}})

It is important to remember that **$gt** is exclusive; documents where `field1` equals 25 will not be included in the result set. If inclusivity is required (meaning 25 should be included), the **$gte** (Greater Than or Equal) operator must be used instead. This distinction is crucial when dealing with precise numerical boundaries or timestamps.

Method 2: Less Than Query (Using $lt)

Conversely, if the objective is to establish an upper limit for the values in a field, the **$lt** operator is employed. This operator selects documents where the value of `field1` is strictly less than the specified value (e.g., 25). This method is perfect for finding data points that fall below a certain threshold.

db.myCollection.find({field1: {$lt:25}})

Similar to **$gt**, **$lt** is exclusive of the boundary value. For scenarios where the boundary value must be included, the **$lte** (Less Than or Equal) operator should be substituted. These single-condition queries form the building blocks for more sophisticated range filtering.

Combining Conditions: AND and OR Logic

While single-condition queries are useful, real-world data retrieval often requires filtering documents that satisfy multiple criteria simultaneously (AND logic) or satisfy one of several criteria (OR logic).

Method 3: Greater Than AND Less Than Query (Range Queries)

The most common complex query involving these operators is defining a range. In MongoDB, when multiple conditions are specified within a single field's query object, they are automatically treated as a logical **AND**. This implicit conjunction makes defining ranges extremely clean and readable.

For example, to find documents where `field1` is between 25 (exclusive) and 32 (exclusive), you combine **$gt** and **$lt** within the same nested structure. This ensures that a document must satisfy **both** conditions to be returned.

db.myCollection.find({field1: {$gt:25, $lt:32}})

This syntax is incredibly powerful for filtering large datasets, such as selecting all orders placed within a specific date range, or identifying user scores that fall within a defined percentile. This implicit **AND** behavior is a hallmark of efficient MongoDB queries, simplifying what would otherwise require an explicit `$and` operator for conditions applied to the same field.

Method 4: Greater Than OR Less Than Query (Disjunction)

When the requirement is to find documents that meet one condition **or** another--meaning the value is either very high **or** very low--you must use the explicit $or logical operator. The $or operator takes an array of query expressions, and a document must match at least one of these expressions to be included in the result set.

In this example, we seek documents where `field1` is strictly greater than 30 **or** strictly less than 20. Notice how each condition is encapsulated within its own complete query object inside the $or array.

db.myCollection.find({ "$or": })

Using $or is essential when filtering against non-contiguous ranges, allowing for highly flexible and targeted data selection that would be impossible with the implicit **AND** behavior of a single field query.

Setting Up the Practical Examples

To solidify our understanding of these comparison operators, we will now apply the four methods demonstrated above to a realistic dataset. We will use a MongoDB collection named `teams`, which stores information about various basketball teams and their current point totals. This collection provides a simple, yet effective, environment for visualizing the effects of **$gt** and **$lt**.

The collection `teams` is populated using the `db.teams.insertOne()` command. Each inserted document contains two primary fields: `team` (a string representing the name) and `points` (a number representing the total score). Before running any queries, the collection must be initialized with the following data:

db.teams.insertOne({team: "Mavs", points: 31})
db.teams.insertOne({team: "Spurs", points: 22})
db.teams.insertOne({team: "Rockets", points: 19})
db.teams.insertOne({team: "Warriors", points: 26})
db.teams.insertOne({team: "Cavs", points: 33})

The complete dataset contains the following point totals: 31, 22, 19, 26, and 33. As we proceed through the examples, pay close attention to which documents are included and which are excluded based on the strict or inclusive nature of the comparison operators used. This hands-on approach illustrates the precision achieved by correctly formulating MongoDB queries.

Example 1: Filtering with $gt (Greater Than)

Our first practical example demonstrates a strict lower bound query using **$gt**. Our goal is to isolate all teams that have scored more than 25 points. This is a common requirement when identifying high-performing data points that exceed a specific benchmark.

The query syntax targets the `points` field and nests the **$gt** operator with the threshold value of 25. Note that because **$gt** is strictly greater than, any team with exactly 25 points would be excluded. The request sent to the MongoDB server is:

db.teams.find({points: {$gt:25}})

Executing this query against the `teams` collection yields the following three documents:

{ _id: ObjectId("6203e4a91e95a9885e1e764f"),
team: 'Mavs',
points: 31 }
{ _id: ObjectId("6203e4a91e95a9885e1e7652"),
team: 'Warriors',
points: 26 }
{ _id: ObjectId("6203e4a91e95a9885e1e7653"),
team: 'Cavs',
points: 33 }

As confirmed by the results, the teams with 31, 26, and 33 points are included. The Spurs (22 points) and Rockets (19 points) are excluded because their scores do not meet the criteria set by the **$gt** 25 condition. This example clearly illustrates how the comparison operators provide precise control over the lower boundary of the result set.

Example 2: Filtering with $lt (Less Than)

To demonstrate the inverse operation, we use the **$lt** operator to establish a strict upper bound. Suppose we want to identify teams that scored poorly, specifically those with a point total strictly less than 25 points. This is useful for identifying outliers or data points that fall below a desired performance metric.

The query syntax uses **$lt** against the `points` field with the value 25. Since **$lt** is exclusive, any document with 25 points would be omitted. We are seeking documents where the score falls into the range of negative infinity up to, but not including, 25.

db.teams.find({points: {$lt:25}})

Upon execution, the system retrieves only two teams:

{ _id: ObjectId("6203e4a91e95a9885e1e7650"),
team: 'Spurs',
points: 22 }
{ _id: ObjectId("6203e4a91e95a9885e1e7651"),
team: 'Rockets',
points: 19 }

The results confirm that only teams scoring 22 and 19 points satisfy the condition. The higher-scoring teams (31, 26, 33) are excluded. This showcases how the **$lt** operator effectively prunes

the dataset to focus on values beneath a specified ceiling. If we had used **$lte**, a team scoring exactly 25 points would have been included.

Example 3: Defining Ranges with Implicit AND Logic

One of the most powerful features of MongoDB queries is the ease with which you can define continuous ranges. By combining **$gt** and **$lt** within the same field definition, MongoDB automatically applies a logical AND condition, selecting only documents that satisfy both criteria. This approach is highly optimized for retrieving data within a specific numerical corridor.

In this scenario, we aim to find "mid-range" teams: those whose points are strictly greater than 25 **AND** strictly less than 32. The query object beautifully illustrates MongoDB's concise syntax for range queries:

db.teams.find({points: {$gt:25, $lt:32}})

The system is instructed to exclude any scores of 25 or less, and 32 or more. This leaves only scores 26 through 31. The resulting dataset confirms this focused retrieval:

{ _id: ObjectId("6203e4a91e95a9885e1e764f"),
team: 'Mavs',
points: 31 }
{ _id: ObjectId("6203e4a91e95a9885e1e7652"),
team: 'Warriors',
points: 26 }

The Mavericks (31 points) and the Warriors (26 points) are returned, as both scores fall between the exclusive bounds of 25 and 32. It is crucial to internalize that when **$gt** and **$lt** (or their inclusive counterparts) are applied to the same field, the **AND** relationship is assumed, resulting in a single, continuous range filter. Had we tried to filter by `{points: {$gt: 32}, points: {$lt: 25}}`, the query would fail or return zero documents, as a document cannot simultaneously have points greater than 32 and less than 25.

Example 4: Handling Non-Contiguous Ranges with $or

When filtering requires selecting documents from two entirely separate, non-overlapping ranges (a disjunction), we must utilize the explicit $or logical operator. The $or operator is crucial for finding outliers--data points that are either significantly above a high threshold or significantly below a low threshold.

For this example, we want to retrieve teams that are either top performers (points strictly greater than 30) **OR** bottom performers (points strictly less than 20). Unlike the implicit **AND** in Example 3, we must define two distinct query objects and pass them as elements within the $or array.

db.teams.find({ "$or": })

The resulting documents satisfy one or both of these conditions:

{ _id: ObjectId("6203e4a91e95a9885e1e764f"),
team: 'Mavs',
points: 31 }
{ _id: ObjectId("6203e4a91e95a9885e1e7651"),
team: 'Rockets',
points: 19 }
{ _id: ObjectId("6203e4a91e95a9885e1e7653"),
team: 'Cavs',
points: 33 }

The output includes the Mavericks (31) and the Cavs (33), both of which are greater than 30, and the Rockets (19), which is less than 20. The Spurs (22) and Warriors (26) are excluded because their scores fall within the intermediate range (20 to 30, inclusive). This method is indispensable for complex filtering where data points outside a primary range need to be selected.

Advanced Considerations and Best Practices

While we primarily used numerical examples, **$gt** and **$lt** are highly versatile. They can be applied effectively to other BSON data types, most notably dates and strings. When applied to dates, these operators allow for efficient time-series querying, such as finding all events occurring before or after a specific timestamp. When applied to strings, the comparison relies on the standard lexicographical order (alphabetical sorting). For example, `{$gt: "M"}` would return all strings starting with N, O, P, and so on.

For optimal performance of range queries, especially on large collections, it is strongly recommended that the fields being queried (e.g., the `points` field) be indexed. Without an appropriate index, MongoDB must perform a collection scan, which can severely degrade query speed as the dataset grows. A single-field index on `{ points: 1 }` would drastically improve the execution time for all the **$gt** and **$lt** queries discussed here.

Finally, always remember the distinction between the exclusive operators (**$gt** and **$lt**) and their inclusive counterparts (**$gte** and **$lte**). A common mistake in query formulation is using an

exclusive operator when the boundary value itself must be included, leading to subtle data omission errors that can impact analysis or application logic. Choosing the correct comparison operators is fundamental to achieving accurate results in your MongoDB queries.

Conclusion

Mastering the **$gt** and **$lt** comparison operators is crucial for effective data manipulation in MongoDB. Whether defining simple thresholds, complex data ranges using implicit **AND** logic, or selecting disparate data points using the explicit $or operator, these tools provide the precision necessary for targeted data retrieval. By understanding the core syntax and the difference between exclusive and inclusive bounds, developers can write clean, efficient, and powerful MongoDB queries that scale with their applications.

To further enhance your skills, we recommend exploring additional MongoDB tutorials covering other common operations and advanced query patterns: