How to Use the MongoDB $or Operator to Simplify Queries

Authored by

stats writer

November 30, 2025

The $or operator is a fundamental component of the MongoDB query language, offering immense flexibility when searching data. It enables users to specify an array of conditions, ensuring that a Document is returned if it satisfies at least one of those criteria. This logical disjunction is essential for constructing complex, highly specific queries that need to filter data based on diverse fields or varying value ranges within a single execution block. Mastering the use of **$or** drastically improves a developer's ability to interact with and retrieve information efficiently from their collections.

Unlike simple equality checks, the **$or** operator allows for the seamless combination of various comparison operators (like **$gt**, **$lt**, **$eq**, etc.) across different fields. This powerful capability means that you do not have to run multiple separate queries and merge the results manually; MongoDB handles the logical union internally. Understanding its structure and practical implementation is key to unlocking advanced data retrieval patterns in non-relational database environments.

You can use the **$or** operator in MongoDB to query for documents that meet one of multiple criteria specified within an array. This operator is crucial for achieving logical disjunction in your filtering logic, operating as a logical OR across the provided conditional expressions.

The **$or** operator requires an array of expressions, where each expression is an independent query condition. This operator adheres to the following basic syntax structure, which must be implemented at the top level of the query document:

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

This specific example instructs MongoDB to search the collection named myCollection and return all documents where **field1** holds the exact value "hello" *or* where **field2** holds a value that is greater than or equal to 10. The condition evaluates to true if either criterion is met, demonstrating the operator's ability to handle diverse field comparisons simultaneously.

Setting Up the Sample Data Collection

To practically demonstrate the capabilities of the **$or** operator, we will use a sample collection named teams. This collection simulates simple statistical data, allowing us to build meaningful queries based on team names, total points, and rebounds. We will insert five documents into this collection, which will serve as our dataset for the following examples.

The initialization of the teams collection involves using the `insertOne()` method for each record, ensuring we have a diverse set of values across the three primary fields: `team` (string), `points`

(number), and `rebounds` (number). This small dataset is ideal for manually verifying the results of our logical OR operations and understanding precisely which documents are matched by the criteria.

The following commands establish the initial state of our teams collection in the MongoDB shell:

```
db.teams.insertOne({team: "Mavs", points: 30, rebounds: 8})
db.teams.insertOne({team: "Mavs", points: 35, rebounds: 12})
db.teams.insertOne({team: "Spurs", points: 20, rebounds: 7})
db.teams.insertOne({team: "Spurs", points: 25, rebounds: 5})
db.teams.insertOne({team: "Spurs", points: 23, rebounds: 9})
```

Example 1: Combining Two Distinct Fields (Equality and Comparison)

Our first example illustrates a typical use case for **$or**: retrieving documents that satisfy one of two conditions involving two distinct fields. We aim to find all documents in the teams collection where either the team name is exactly "Spurs" *or* the accumulated points are significantly high (greater than or equal to 31). This combination demonstrates how **$or** facilitates complex filtering based on categorical data and numerical thresholds simultaneously.

The structure requires providing the **$or** operator with an array containing two separate query documents. The first document specifies the equality condition (`{"team": "Spurs"}`), and the second document uses the $gte operator (`{"points": {$gte: 31}}`) to check the numerical threshold. Both conditions are treated independently, and if a document matches either one, it is included in the final result set produced by the query.

Execute the following query in the MongoDB shell to retrieve the desired data:

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

Analyzing the Result Set from Example 1

The execution of the preceding query returns four distinct documents from the collection. It is crucial to examine the results to confirm that every returned document satisfies at least one of the two specified criteria: `team == "Spurs"` OR `points >= 31`. This verification confirms the correct logical application of the **$or** operator, providing a logical union of the two condition sets.

The first returned document is the 'Mavs' entry with 35 points; it satisfies the second condition (35

is greater than or equal to 31). The remaining three documents are all 'Spurs' entries, satisfying the first condition. The 'Mavs' entry with 30 points is notably excluded because it is neither a "Spurs" team nor does it meet the points threshold (30 < 31). The $or operator thus efficiently filters out documents that fail all provided conditions.

The resulting documents are:

{ _id: ObjectId("62018750fd435937399d6b6f"),
team: 'Mavs',
points: 35,
rebounds: 12 }
{ _id: ObjectId("62018750fd435937399d6b70"),
team: 'Spurs',
points: 20,
rebounds: 7 }
{ _id: ObjectId("62018750fd435937399d6b71"),
team: 'Spurs',
points: 25,
rebounds: 5 }
{ _id: ObjectId("62018750fd435937399d6b72"),
team: 'Spurs',
points: 23,
rebounds: 9 }

Notice that each document in the output contains "Spurs" in the team field *or* a value greater than or equal to 31 in the points field. This clearly demonstrates the logical union provided by the **$or** operator when combining conditions across multiple fields.

Example 2: Leveraging $or with Multiple Conditions and Operators

The power of the $or operator is not limited to just two criteria; it can handle an arbitrary number of expressions within its array argument. In this expanded example, we seek documents that meet any of three distinct requirements across three separate fields: team equality, a points threshold, or a rebounds threshold. This showcases the operator's versatility in combining diverse filtering logic.

Specifically, we are looking for documents where the team is "Mavs", OR the points are greater than or equal to 25 (using **$gte**), OR the rebounds are strictly less than 8 (using the **$lt** operator). By including three logical tests, we are significantly broadening the scope of the query, making it more likely that a document will satisfy at least one criterion from the provided array of condition objects.

The array passed to **$or** now contains three separate condition objects, each defining one required state. This flexible structure is highly valuable when defining complex business rules for data retrieval in a database like MongoDB:

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

Result Interpretation for Example 2

The execution of the three-condition **$or** query yields four documents. It is important to confirm why each document was included based on the criteria: 1) Team equals "Mavs," 2) Points >= 25, or 3) Rebounds < 8. Note that documents satisfying multiple criteria are still only returned once.

The two "Mavs" documents satisfy criterion 1 (team). The Spurs document with 25 points satisfies criterion 2 (points >= 25) and criterion 3 (5 rebounds < 8). The Spurs document with 20 points satisfies criterion 3 (7 rebounds < 8). Only the Spurs entry with 23 points and 9 rebounds is excluded, as it fails all three tests simultaneously, providing clear validation of the logical disjunction operation.

The final returned result set for this query is:

{ _id: ObjectId("62018750fd435937399d6b6e"),
team: 'Mavs',
points: 30,
rebounds: 8 }
{ _id: ObjectId("62018750fd435937399d6b6f"),
team: 'Mavs',
points: 35,
rebounds: 12 }
{ _id: ObjectId("62018750fd435937399d6b70"),
team: 'Spurs',
points: 20,
rebounds: 7 }
{ _id: ObjectId("62018750fd435937399d6b71"),
team: 'Spurs',
points: 25,
rebounds: 5 }

Each document aligns with the specified conditions, where at least one of the following is true:

The "team" field is equal to "**Mavs**"
The "points" field has a value greater than or equal to **25**
The "rebounds" field has a value less than **8**

Best Practices and Performance Considerations for $or Queries

While the $or operator is functionally powerful, its usage, especially across many fields or very large collections, demands attention to performance optimization. In MongoDB, queries involving **$or** can sometimes be less efficient than those using the implicit logical AND condition or the **$in** operator, depending on the underlying indexing strategy and the nature of the conditions being evaluated.

A crucial best practice is to analyze your filtering requirements. If all conditions within the **$or** array are simple equality checks on the same field (e.g., finding documents where `category` is 'A' OR 'B' OR 'C'), it is highly recommended to use the **$in** operator instead. The **$in** operator is generally optimized for matching multiple values against a single key, leading to significantly faster lookups, particularly when indexes are present and the array of values is small or moderate in size.

For complex cases where **$or** is necessary--such as combining queries across different fields or using various range operators (like **$gte** or **$lt**)--indexing becomes paramount. MongoDB must evaluate each component of the **$or** array separately. If suitable indexes exist for the fields mentioned in the conditions, the query planner can utilize an index intersection or union optimization, dramatically reducing the amount of data scanned and improving query latency.

$or and Effective Indexing Strategies

Indexing is the single most important factor influencing the performance of **$or** queries. When an **$or** query spans multiple fields, MongoDB attempts to use an index on each field involved. If the query planner finds useful indexes for all clauses within the **$or** array, it can merge the results from index scans, a technique optimized for retrieving the final set of matching documents.

To maximize performance, ensure that every field referenced within the **$or** conditions that has high selectivity (many unique values) is indexed. For instance, in Example 1, where we queried on `team` and `points`, having separate indexes on `{team: 1}` and `{points: 1}` would allow MongoDB to efficiently resolve both parts of the condition and combine the results quickly. If one clause within the **$or** array cannot use an index, the entire query may default to a full collection scan, severely impacting performance on large datasets. Always use `.explain("executionStats")` to verify that your indexes are being utilized effectively by the query optimizer.

Conclusion

The $or operator is an indispensable tool in the MongoDB query arsenal, enabling developers to build flexible and robust filtering logic by combining multiple criteria through logical disjunction. We have demonstrated its syntax, its application across different fields and operators, and the crucial considerations for maintaining query performance on large collections by focusing on proper index utilization.

By effectively using **$or**--and judiciously considering alternatives like **$in** for single-field matches--you can ensure your data retrieval operations are both accurate and scalable within your MongoDB environment. Remember to prioritize proper indexing to guarantee efficient execution, especially when combining conditions across multiple, distinct fields.

Note: You can find the complete official documentation for the **$or** function here.

Related MongoDB Query Tutorials

The following tutorials explain how to perform other common operations in MongoDB, including the complementary logical AND operator and handling null value checks, which often work alongside or as alternatives to the **$or** operator:

MongoDB: How to Use the AND Operator in Queries

MongoDB: How to Query for "not null" in Specific Field