Learning MongoDB: Implementing “Like” Queries with Regular Expressions

Authored by

Mohammed looti

November 1, 2025

In conventional SQL databases, the `LIKE` operator serves as the standard mechanism for flexible string matching, allowing developers to execute powerful partial searches against textual data. Conversely, MongoDB, a leading NoSQL document database, achieves this essential functionality using **Regular Expressions** (Regex) applied through the native **$regex** operator. This detailed tutorial provides a structured approach to implementing and mastering the three primary methods necessary to effectively replicate and significantly enhance the capabilities of the SQL `LIKE` clause within the MongoDB query language.

Understanding Regular Expressions in MongoDB

The foundation of pattern matching in MongoDB rests entirely upon the **$regex** operator. Whenever a query necessitates dynamic search capabilities or partial string identification, utilizing standard Regular Expressions (often shortened to Regex) is the definitive approach. This powerful operator enables developers to embed complex patterns directly within the query object supplied to the `find()` method. This contrasts sharply with simple equality comparisons, which require an exact match; Regex grants precise control over the location of the pattern--whether it must be anchored to the start, the end, or appear anywhere within the target field's content.

The inherent flexibility of the **$regex** operator makes it indispensable for developing sophisticated application features, including high-speed, real-time search functionalities, adaptive content filtering mechanisms, and stringent data validation processes. To fully harness this potential, the remainder of this guide meticulously details the specific syntax required to execute searches that mimic SQL's `LIKE` behavior: finding strings that contain a specified pattern (unanchored search), strings that begin with a specific prefix, and strings that conclude with a defined suffix.

Setting Up the Example Dataset

To provide clear, practical demonstrations of these MongoDB query techniques, we will work with a dedicated sample collection. This collection, named teams, contains several documents, each structured to hold data pertaining to a basketball team, specifically recording the team's name, the primary player position, and the associated points scored.

We initialize the collection by inserting the following documents using the standard `insertOne` command. These distinct field values are strategically chosen to allow us to test and verify the different pattern matching methods described in the subsequent sections:

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

Method 1: Finding Documents that Contain a Substring (Unanchored "Like")

This first method is the most frequently utilized, as it directly mirrors the general SQL pattern `LIKE '%string%'`. Its purpose is to locate documents where the specified pattern exists anywhere within the target field's value, without restriction to the beginning or end of the string. In MongoDB, this widespread search capability is achieved by defining the search term as a **Regular Expression** literal, which is conventionally enclosed within forward slashes (`/`).

The generic syntax below illustrates how to construct a query using the **$regex** operator to search a field named name for the general occurrence of 'string':

db.collection.find({name: {$regex : /string/i}})

A crucial component of this structure is the i flag, which is appended directly after the Regex pattern (e.g., `/string/i`). This flag designates the match as case-insensitive, ensuring that the query successfully retrieves results regardless of the capitalization utilized in either the database document or the search pattern provided by the user.

Example 1: Searching for 'avs' within Team Names

Our objective here is to locate all documents where the team field includes the specific substring 'avs'. Based on our sample dataset, we expect this unanchored pattern to successfully match both 'Mavs' and 'Cavs'. We apply this containment logic using the **$regex** operator:

db.teams.find({team: {$regex : /avs/i}})

As anticipated, executing this query successfully retrieves the following two documents, which confirms the effective functionality of an unanchored, case-insensitive Regex search for substrings:

{ _id: ObjectId("618050098ffcfe76d07b1da5"),
team: 'Mavs',
position: 'Guard',
points: 31 }

{ _id: ObjectId("618285361a42e92ac9ccd2c6"),
team: 'Cavs',
position: 'Guard',
points: 33 }

Method 2: Matching Documents that Start with a Specific Prefix (Starting Anchor)

To precisely replicate the SQL pattern `LIKE 'string%'`, which demands that the pattern be fixed solely at the beginning of the string, we utilize the caret symbol (^). This symbol functions as a critical metacharacter within Regular Expressions, asserting the absolute start position of the field value. Utilizing this starting anchor is highly recommended as it makes the search highly specific and, crucially, often allows MongoDB to leverage indexes for vastly improved query performance compared to unanchored searches.

The following syntax demonstrates the required structure. By placing the `^` immediately before the desired pattern, we ensure that the **$regex** query only returns documents where the specified field, name, strictly commences with the sequence 'string':

db.collection.find({name: {$regex : /^string/i}})

Example 2: Searching for Positions Starting with 'gua'

Suppose our primary objective is to retrieve all player documents where the position field starts with the character sequence 'gua'. To enforce this prefix requirement, we must place the starting anchor (^) immediately before 'gua' within the Regex literal:

db.teams.find({position: {$regex : /^gua/i}})

Upon execution, this prefix-anchored query successfully retrieves three documents from our example collection. Since 'Guard' starts with 'gua', all three team entries assigned to that position are returned, demonstrating the precise nature of the starting anchor:

{ _id: ObjectId("618050098ffcfe76d07b1da5"),
team: 'Mavs',
position: 'Guard',
points: 31 }

{ _id: ObjectId("6180504e8ffcfe76d07b1da7"),
team: 'Spurs',
position: 'Guard',
points: 22 }

{ _id: ObjectId("618285361a42e92ac9ccd2c6"),
team: 'Cavs',

position: 'Guard',
points: 33 }

Method 3: Matching Documents that End with a Specific Suffix (Ending Anchor)

To achieve string matching equivalent to the SQL pattern `LIKE '%string'`, which requires the match to occur only at the end of the text, we use the dollar sign symbol (`$`) within our **Regular Expressions**. The dollar sign functions as a vital anchor, asserting that the pattern must terminate exactly at the end of the string value. This configuration ensures that the retrieved pattern constitutes the absolute final sequence of characters contained within the field.

The structure necessary for suffix matching involves placing the desired pattern immediately followed by the `$` anchor within the Regex literal provided to the **$regex** operator:

db.collection.find({name: {$regex : /string$/i}})

Example 3: Searching for Positions Ending with 'ward'

In this final example, we construct a query targeting the position field, specifically seeking documents that conclude with the suffix 'ward'. This precise requirement is intended to isolate any documents corresponding to the 'Forward' position within our dataset.

db.teams.find({position: {$regex : /ward$/i}})

The execution of this query successfully isolates the single relevant document, which pertains to the 'Warriors' team. This confirms that the ending anchor effectively limits the match to only those field values that conclude precisely with the specified suffix:

{ _id: ObjectId("618050808ffcfe76d07b1dab"),
team: 'Warriors',
position: 'Forward',
points: 26 }

Optimizing $regex Queries for Performance and Indexing

Although the **$regex** operator offers tremendous power and flexibility for string matching, developers must carefully consider its impact on performance, especially when executing queries against large collections in a production MongoDB environment. The primary concern arises with

unanchored regex queries (such as those demonstrated in Method 1), which inherently require a full collection scan, leading to high computational expense and drastically increased latency. Understanding indexing limitations is vital for maintaining scalable application performance.

To ensure optimal query performance and efficient resource utilization, developers should strictly adhere to the following best practices concerning index utilization and **Regular Expressions**:

Prefix Anchoring is Crucial: MongoDB is only able to effectively utilize traditional indexes for regex searches if the pattern is defined as a prefix expression. This mandates anchoring the pattern explicitly to the beginning of the string using the caret (^) metacharacter.
Index Limitations and Flags: Standard indexes generally cannot be leveraged if the **$regex** query includes the ending anchor `$` or, critically, the `i` (case-insensitive) flag. To index case-insensitive searches, specialized solutions such as text indexes or indexes configured with collation must be employed.
Strategic Search Limitation: Global, unanchored searches should be minimized, particularly on high-volume, frequently accessed fields. If unanchored searches are unavoidable, mitigate the performance impact by combining the regex criteria with other highly indexed fields (for example, filtering by a regex pattern only within a narrowly defined date range).

In conclusion, when designing string pattern searches, the starting anchor (^) should always be prioritized whenever the search logic permits. This simple step allows MongoDB to harness existing indexes, resulting in significantly reduced query execution time and improved scalability. For comprehensive details regarding advanced syntax, performance nuances, and the utilization of collation for case-insensitive indexing, always consult the official MongoDB documentation site.