How to Group and Count Documents in MongoDB: A Simple Guide

Authored by

stats writer

December 2, 2025

MongoDB is a powerful NoSQL database that offers sophisticated tools for data analysis and reporting, far beyond simple CRUD operations. One of the most common analytical tasks is summarizing data--specifically, grouping documents based on a shared attribute and calculating metrics for those groups, such as the total count. This process is handled efficiently through the **Aggregation Pipeline**, a multi-stage data processing framework designed for complex queries.

When working with large datasets, developers frequently need to understand the distribution of values within a specific field. For instance, determining how many users belong to each region or counting the inventory items by category. **MongoDB**'s aggregation framework, particularly the **$group** stage, provides the functionality necessary to perform these counts and classifications directly within the database server, minimizing data transfer overhead and maximizing performance. This tutorial will detail the exact syntax and provide practical examples for grouping and counting documents.

Essential Syntax for Grouping and Counting

To perform a standard group-by and count operation in **MongoDB**, you must invoke the `.aggregate()` method on your target collection and pass an array containing the aggregation stages. The simplest structure involves just one crucial stage: the **$group** operator. This stage requires two main components: the `_id` field, which defines the grouping key, and the new fields being calculated (in this case, the count).

The syntax below illustrates how to use the **$group** stage. Notice that the value assigned to `_id` must be prefixed with a dollar sign (`$`) followed by the name of the field you wish to group by. The output field, named `count` in this example, uses the **$sum** accumulator with a constant value of 1, ensuring every document contributes exactly one unit to its respective group's total.

db.collection.aggregate()

It is important to remember that `field_name`, contained within the `_id` definition, must be replaced with the actual name of the document field you intend to use as the grouping key. The result of this operation is a set of documents, each containing the unique grouping key (in the `_id` field) and the corresponding document count (in the `count` field). This powerful mechanism forms the basis of many reporting queries in **MongoDB**.

Setting Up the Sample Dataset

To effectively demonstrate the grouping and counting functionality, we will utilize a sample dataset representing various sports teams and their player statistics. We will work with a collection named `teams`, which contains fields such as `team`, `position`, and `points`. This dataset allows us to

explore how to count occurrences based on categorical data like player position.

Below are the five documents inserted into the `teams` collection. Analyze the data to anticipate the output of our aggregation queries. For instance, observe how many players are listed under the 'Guard' position compared to 'Center' or 'Forward'.

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})

This small but representative collection will serve as the foundation for the subsequent examples, allowing us to immediately see the practical impact of the **$group** stage on real-world data distribution problems.

Example 1: Basic Grouping by a Field

Our first example focuses solely on using the **Aggregation Pipeline** to group the documents in the `teams` collection based on the `position` field. This is the most straightforward application of the grouping functionality, designed to calculate the frequency of each unique position present in the collection.

We pass the `$position` field reference to the `_id` parameter within the **$group** stage. The `_id` field serves as the unique identifier for the resulting aggregated document. Concurrently, we define a new field called `count`, accumulating the total number of documents that fall into that specific position category using `{$sum: 1}`.

db.teams.aggregate()

Executing the code above yields the following aggregated output. Notice that the output documents are concise, listing only the grouping key (`_id`) and the calculated count, summarizing the underlying data distribution effectively.

{ _id: 'Forward', count: 1 }
{ _id: 'Guard', count: 3 }
{ _id: 'Center', count: 1 }

The result clearly indicates the frequency of each player position within our sample collection:

The position 'Forward' occurs **1** time.
The position 'Guard' occurs **3** times.
The position 'Center' occurs **1** time.

Understanding Accumulators and $sum

While this article focuses on counting documents using `{$sum: 1}`, it is vital to understand the role of **accumulators** within the **Aggregation Pipeline**. Accumulators are essential operators used within the **$group** stage to perform calculations across the documents belonging to a group. Beyond counting, aggregations can calculate averages, maximums, minimums, and even push values into arrays.

The **$sum** operator is arguably the most frequently used accumulator. When calculating a count, we simply sum a constant value of 1 for every document encountered. However, if we wanted to calculate the total points scored by all players in each position, we would replace `1` with `$points`, thereby summing the numerical value of that field across the group instead of just counting the documents.

This flexibility allows for complex metric calculation within a single aggregation query. Whether you are generating a simple frequency distribution or calculating financial totals for different regions, the combination of `_id` (the grouping key) and one or more accumulators provides the analytical power needed for robust data querying in **MongoDB**.

Example 2: Combining Grouping with $sort for Ordered Results

In many reporting scenarios, merely obtaining the counts is insufficient; the results must be ordered, often to highlight the most or least frequent categories. This requires adding a second stage to the **Aggregation Pipeline**: the **$sort** stage. The `$sort` stage must follow the **$group** stage, as it operates on the newly generated aggregated documents.

We can use the following syntax to first count the occurrences of each position and then automatically sort the results in **ascending order** based on the resulting `count` field. Ascending order is designated by the value `1` passed to the sort key. This is useful for identifying the least common categories within the dataset.

db.teams.aggregate()

After the grouping occurs, the documents are processed by the **$sort** stage. This returns the following results, ordered from the lowest count to the highest:

{ _id: 'Forward', count: 1 }

{ _id: 'Center', count: 1 }
{ _id: 'Guard', count: 3 }

Understanding Sorting Logic (Ascending vs. Descending)

The sorting direction is determined by the numerical value assigned to the field within the **$sort** stage. A value of `1` specifies ascending order (A to Z, 0 to 9), while a value of `-1` specifies descending order (Z to A, 9 to 0). Descending order is typically preferred when trying to find the most frequent categories or the 'top N' results.

To demonstrate descending order, we simply modify the sort stage by changing the value associated with the `count` field from `1` to `-1`. This small change flips the output, providing immediate visibility into the groups with the highest frequency, which in our case is the 'Guard' position.

db.teams.aggregate()

The following output shows the results sorted in descending order of the count. This structure is often used as the basis for further processing, such as limiting the results to the top 5 groups using the `$limit` stage.

{ _id: 'Guard', count: 3 }
{ _id: 'Forward', count: 1 }
{ _id: 'Center', count: 1 }

Further Aggregation Opportunities

While grouping and counting form the foundation of analytical queries, the **$sort** stage can be combined with many other powerful stages. For instance, to retrieve the top two player positions by count, you would simply chain a `{$limit: 2}` stage immediately after the descending `$sort` stage.

Furthermore, before grouping, you might need to filter the data using the `$match` stage (similar to a SQL WHERE clause) to only include documents that meet certain criteria, such as players scoring more than 20 points. By carefully sequencing these stages--matching, grouping, calculating, and sorting--you can construct highly specific and efficient analytical queries tailored to complex business requirements.

For developers seeking deeper insights into the intricacies of grouping and aggregation, the official **$sum** documentation provides comprehensive details on all available accumulators and

aggregation expressions.