Learning MongoDB: Grouping Data by Multiple Fields

Authored by

Mohammed looti

November 1, 2025

Mastering the aggregation pipeline is fundamental for performing sophisticated data analysis and transformation within MongoDB. Unlike traditional relational databases that use the standard SQL **GROUP BY** clause, MongoDB achieves this functionality using the powerful **$group** stage. A very common requirement in reporting is grouping documents based on multiple criteria simultaneously, which allows for highly specific data summarization across combined fields. This technique is essential for deriving meaningful metrics from raw NoSQL data structures.

To successfully group documents by multiple fields and concurrently apply essential data aggregation operations, developers must leverage the **$group** stage within the aggregation framework. The key to multi-field grouping lies in defining a composite key in the required **_id** field. This structure ensures that documents are grouped only when the values of all specified fields match exactly. The following general syntax outlines how to define this composite key and pair it with an appropriate accumulator function:

db.collection.aggregate()

For a detailed, practical demonstration of this functionality, we will utilize a sample collection named `teams`. This collection contains statistics related to players, tracking crucial data points such as their team affiliation, specific position, and the total points they have scored. The documents provided below represent the foundational dataset that we will be using throughout the subsequent aggregation examples:

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

Grouping and Counting Documents by Multiple Fields

Our initial analytical task requires us to determine the frequency of players belonging to every unique combination of 'team' and 'position'. This is a foundational step in understanding the distribution of roles within the dataset. To achieve this count, we must employ the $group stage as the sole stage in our pipeline.

The crucial design decision here is defining the **_id** field. Instead of referencing a single field, we define it as an object containing references to both `$team` and `$position`. This composite structure forces MongoDB to treat the pair of values as a single, unique grouping key. Furthermore, we introduce the `count` field, which utilizes the **$sum: 1** accumulator. By summing the value 1 for every document that flows into a group, we effectively count the number of

documents associated with that unique composite key. This methodology mirrors the functionality of a COUNT(*) operation combined with a **GROUP BY** clause in standard relational database systems.

db.teams.aggregate()

Upon executing the aggregation pipeline defined above, we receive a structured output that clearly maps each unique team and position combination to its corresponding document frequency. This immediately reveals, for example, that the 'Mavs' team has two 'Guard' players documented, while all other combinations in this small sample dataset only contain one instance.

{ _id: { team: 'Rockets', position: 'Forward' }, count: 1 }
{ _id: { team: 'Mavs', position: 'Guard' }, count: 2 }
{ _id: { team: 'Mavs', position: 'Forward' }, count: 1 }
{ _id: { team: 'Rockets', position: 'Guard' }, count: 1 }

Performing Advanced Aggregate Calculations: Summing Data

The utility of the $group stage extends far beyond simple document counting. We can leverage other powerful accumulator operators to calculate comprehensive metrics, offering deeper insight into collective performance based on our composite grouping criteria. A common requirement in performance analysis is calculating the total points scored by players for each specific team and position pairing.

To achieve this specific calculation, we maintain the same composite key definition in the **_id** field, ensuring accurate segmentation by team and position. However, instead of counting documents, we define a new output field, `sumPoints`. Within this field, we apply the specialized $sum operator directly to the `$points` field from the original source documents. This tells MongoDB to accumulate the values of the `points` field for every document that belongs to that specific group.

db.teams.aggregate()

The resulting output documents now reflect the total points aggregated for each unique combination, moving beyond simple counts to provide valuable summary statistics derived directly from the raw document data. This immediate insight is crucial for performance reviews and strategic decision-making.

{ _id: { team: 'Rockets', position: 'Forward' }, sumPoints: 33 }
{ _id: { team: 'Mavs', position: 'Guard' }, sumPoints: 53 }
{ _id: { team: 'Mavs', position: 'Forward' }, sumPoints: 19 }

{ _id: { team: 'Rockets', position: 'Guard' }, sumPoints: 26 }

Interpreting these results allows us to quickly quantify performance across different groups:

The total points scored by players designated as 'Guard' for the 'Mavs' team is the highest metric in this sample, totaling **53** points.
Conversely, the 'Forward' players on the 'Mavs' team contributed **19** points, representing the lowest total in the aggregated dataset.
This form of aggregation provides immediate, actionable summary statistics that are impossible to derive efficiently without the aggregation pipeline.

Chaining Aggregation Stages: Grouping and Sorting Results

While grouping and calculating metrics are vital, the resulting data often requires presentation in a specific, ordered sequence to maximize its usefulness. Fortunately, the sequential nature of the aggregation framework allows for easy chaining of operations. We can simply add a **$sort** stage immediately following the $group stage. This enables us to order the resulting aggregated documents based on the newly calculated fields, such as `sumPoints`.

To demonstrate this, the following command first groups the documents and calculates the summed points using the $sum operator. Subsequently, it uses the $sort stage to order the results in **ascending order**. In MongoDB, ascending order is designated by the numeric value `1` applied to the target field, `sumPoints`. This approach is helpful for identifying groups that are underperforming or contributing the least to the metric.

db.teams.aggregate()

When the results are sorted in ascending order based on `sumPoints`, the groups with the lowest summed totals are prominently displayed first. This provides an immediate, ordered ranking of the combined criteria:

{ _id: { team: 'Mavs', position: 'Forward' }, sumPoints: 19 }
{ _id: { team: 'Rockets', position: 'Guard' }, sumPoints: 26 }
{ _id: { team: 'Rockets', position: 'Forward' }, sumPoints: 33 }
{ _id: { team: 'Mavs', position: 'Guard' }, sumPoints: 53 }

Sorting Results in Descending Order for Top Performance Analysis

In contrast to ascending order, data analysts often prefer to see the highest values first when evaluating performance metrics. To achieve this reverse ordering--displaying the highest point

totals at the beginning of the result set--we simply modify the $sort stage by using the value **-1**. The negative one parameter indicates a descending sort direction. This presentation style is invaluable for quickly identifying leading groups or top performers based on the calculated aggregated metric.

db.teams.aggregate()

The resulting documents are now sorted in **descending order** based on the `sumPoints` field. The output clearly starts with the group that generated the maximum number of points, providing immediate visibility into the most productive team/position combinations:

{ _id: { team: 'Mavs', position: 'Guard' }, sumPoints: 53 }
{ _id: { team: 'Rockets', position: 'Forward' }, sumPoints: 33 }
{ _id: { team: 'Rockets', position: 'Guard' }, sumPoints: 26 }
{ _id: { team: 'Mavs', position: 'Forward' }, sumPoints: 19 }

Summary of Multi-Field Grouping Techniques

Grouping data by multiple fields is an indispensable skill for anyone performing complex reporting and in-depth data analysis within MongoDB. The entire methodology hinges on defining a precise composite key within the mandatory **_id** field of the $group stage. This technique allows developers to segment their data with granularity and apply powerful accumulators such as $sum, $avg, or $max.

Furthermore, the true power of the aggregation pipeline is revealed when chaining multiple operations. By following the aggregation step with stages like $sort, results can be flexibly ordered and filtered, transforming raw output into actionable business intelligence.

For developers seeking mastery over the aggregation framework, we strongly recommend consulting the official $group documentation. This resource provides comprehensive details on all available operators and their specific capabilities, ensuring you can tailor complex queries to meet any data requirement.

Note: The documentation for the **$group** stage offers extensive examples for complex data modeling scenarios.