How to Group Data by Date in MongoDB: A Simple Guide

Authored by

stats writer

November 30, 2025

The ability to group documents based on date fields is a cornerstone of effective data analysis. In MongoDB, this functionality is robustly handled through the powerful Aggregation Pipeline. Grouping by date allows users to consolidate complex time-series data, making it straightforward to identify crucial patterns, recurring trends, or seasonal variations within a dataset.

This technique is vital for business intelligence (BI) tasks, such as tracking daily website traffic, monitoring sales performance over monthly cycles, or analyzing user activity across specific time frames. By transforming raw date objects into structured, comparable groupings, developers and analysts can efficiently manage and derive meaningful insights from large volumes of documents. The efficiency gained by using date grouping is especially critical when dealing with large-scale collections where manual sorting and counting would be computationally prohibitive.

Understanding the MongoDB Aggregation Pipeline

Grouping documents in MongoDB is achieved using the Aggregation Pipeline. This pipeline operates much like a set of processing stages where documents flow through sequential operations, transforming the data at each step. The primary operator utilized for grouping is $group, which collects input documents and groups them based on a specified identifier expression, known as the `_id` field.

To group by date, we must instruct MongoDB to extract only the necessary date component (e.g., Year, Month, Day) from the full datetime object stored in the documents. If we simply used the raw datetime field, MongoDB would treat every document with a unique timestamp (down to the millisecond) as a separate group. Therefore, we utilize specialized date aggregation operators, such as $dateToString, within the `_id` definition to format the date correctly for grouping.

This transformation step is crucial for achieving accurate aggregation results, allowing us to aggregate statistics--such as counts or sums--for all records that fall within the same calendar day, month, or year, regardless of the specific time of entry. The first stage in our pipeline will almost always be the $group stage when performing date-based analysis.

Constructing the Basic Date Grouping Query

The syntax for grouping documents by date involves specifying the $group stage and defining the grouping key using `$dateToString`. This operator accepts a format string that dictates how the date object will be rendered, effectively creating the shared grouping key.

You can use the following syntax to group documents by date in MongoDB:

db.collection.aggregate()

Note that day is the name of the date field that we'd like to group by in this example.

In this construction, the `$group` stage defines the accumulator fields. The `_id` field specifies the grouping key, generated by `$dateToString`. The format string `"%Y-%m-%d"` ensures that the output key for grouping only includes the year, month, and day. We then define a new field, `count`, which uses the $sum accumulator set to `1`. By setting $sum to `1`, we instruct the pipeline to count every document that flows into that specific date group, effectively calculating the total number of records per day.

Preparing the Sample Dataset for Analysis

To effectively demonstrate date grouping, we will work with a sample MongoDB collection named sales. This collection stores simple documents tracking transactions, each containing a date field (`day`) and a numerical value (`amount`). While our resulting aggregations will show counts greater than one for certain dates, the following insertion commands illustrate the structure of the data records:

The following examples show how to use this syntax with a collection sales with the following documents:

db.sales.insertOne({day: new Date("2020-01-20"), amount: 40})
db.sales.insertOne({day: new Date("2020-01-21"), amount: 32})
db.sales.insertOne({day: new Date("2020-01-22"), amount: 19})
db.sales.insertOne({day: new Date("2020-01-23"), amount: 29})
db.sales.insertOne({day: new Date("2020-01-24"), amount: 35})

In a real-world scenario, the `sales` collection would contain hundreds or thousands of documents. The critical element here is the `day` field, which stores the date as a BSON Date object. This is the field targeted by the `$dateToString` operator in our aggregation query, ensuring that we can accurately group documents together based solely on the calendar date, ignoring the time component.

Understanding the input data format is crucial because the aggregation pipeline processes these documents sequentially. If the `day` field were stored as a string instead of a BSON Date, we would have to use different string manipulation operators to extract the grouping key, potentially complicating the query and affecting performance. Using native Date objects whenever possible is highly recommended for time-series aggregation tasks.

Example 1: Grouping Documents by Date and Calculating Counts

The most fundamental use case for date grouping is determining the frequency of events or

records per day. This aggregation counts how many documents exist for each unique date present in the `sales` collection. This is achieved entirely within the single $group stage, utilizing the $sum accumulator to calculate the total count.

We can use the following code to count the number of documents, grouped by date:

db.sales.aggregate()

This query processes the entire collection, aggregates documents by the string representation of the date, and produces a summary result set. Each resulting document represents a unique date found in the collection, along with the calculated frequency:

This query returns the following results:

{ _id: '2020-01-20', count: 2 }
{ _id: '2020-01-22', count: 1 }
{ _id: '2020-01-21', count: 2 }

These results clearly show the distribution of documents across the specific dates. For instance, the result indicates that on January 20, 2020, there were two documents (or sales records) created, while on January 22, 2020, only one record was created.

From the results we can see:

The date 2020-01-20 occurs **2** times.
The date 2020-01-22 occurs **1** time.
The date 2020-01-21 occurs **2** times.

Example 2: Enhancing Analysis with Sorting

While grouping by date provides valuable counts, it is often necessary to order these aggregated results to identify peak activity days or periods of low volume. This is achieved by adding a second stage to the pipeline: the $sort operator. The $sort stage is applied after the $group stage has already calculated the counts, allowing us to order the output documents based on the newly created `count` field.

Sorting can be performed in two primary directions: **ascending** (smallest to largest, specified by `1`) or **descending** (largest to smallest, specified by `-1`). Sorting by count in ascending order is useful for quickly identifying dates with the least activity.

We can use the following code to count the number of documents, grouped by date, and sort the

results based on count **ascending**:

db.sales.aggregate()

In this pipeline, the first stage performs the grouping and counting, and the second stage takes the resulting aggregated documents and orders them by the `count` field in ascending order. This places the least frequent dates at the beginning of the output:

This query returns the following results:

{ _id: '2020-01-22', count: 1 }
{ _id: '2020-01-20', count: 2 }
{ _id: '2020-01-21', count: 2 }

Sorting in Descending Order

Conversely, sorting in descending order is typically used to highlight the most active dates, allowing immediate identification of peak performance or high-traffic periods. By changing the sort parameter from `1` to `-1`, the order is reversed, placing the largest counts first.

The following query demonstrates how to sort the grouped results by `count` in descending order:

db.sales.aggregate()

By using `-1` for the `count` field in the $sort stage, we prioritize the dates with the highest number of records. This is invaluable for immediate trend spotting or error monitoring, where high volumes might indicate either success or anomaly.

This query returns the following results:

{ _id: '2020-01-20', count: 2 }
{ _id: '2020-01-21', count: 2 }
{ _id: '2020-01-22', count: 1 }

Note: You can find the complete documentation for the **sort** function on the official MongoDB website, which details options for multi-field sorting and index utilization.

Advanced Date Grouping: Summing Numerical Fields

While counting documents is the simplest form of date aggregation, the $group stage is much

more versatile. Instead of merely counting documents using `$sum: 1`, we can calculate the total sum of a numerical field for all documents belonging to a particular date group. This is typically used for calculating total sales, total revenue, or total duration per day.

In our `sales` collection example, documents contain an `amount` field. We can easily modify our aggregation to calculate the total sales amount for each day, rather than just the number of transactions:

db.sales.aggregate()

By replacing `1` with `"$amount"` in the $sum accumulator, we instruct the pipeline to sum the value of the `amount` field from every document belonging to the group defined by `_id`. This results in a comprehensive financial summary, showing the total revenue generated on each specific date.

Further Date Aggregation Operations

The techniques discussed here form the foundation for complex date-based queries. MongoDB offers numerous other date operators and aggregation functions that allow for grouping by week (using `%W`), by month (using `%Y-%m`), or even by hour. Furthermore, the aggregation pipeline supports multiple simultaneous aggregations within a single $group stage, allowing analysts to calculate minimums, maximums, and averages alongside the sum and count.

For advanced requirements, operators like `$bucket` or `$facet` can be combined with date grouping to create time-series buckets or generate multi-faceted reports based on time and other document attributes. Mastering the core principles of using `$group` and `$dateToString`, however, is the essential first step toward generating complex, insightful reports from your time-stamped data.

The following tutorials explain how to perform other common operations in MongoDB: