How to Easily Join Fields from Two MongoDB Collections

Authored by

stats writer

November 30, 2025

When dealing with data manipulation in MongoDB, the term "joining fields" can refer to two distinct but equally important operations: linking data across multiple collections (the traditional database join), or combining the values of two or more fields within a single document (string concatenation). While **MongoDB** is a NoSQL document database, it provides powerful tools to handle relational data requirements. For collection joins, the dedicated operator is **$lookup**, which simulates a left outer join, allowing developers to pull related data from one collection into another, greatly enhancing data retrieval efficiency and structuring capabilities for complex queries.

However, often the requirement is simpler: to merge text data from existing fields--such as first name and last name, or team name and conference--into a new, combined field within the same document. This operation is achieved using the $concat operator, executed within the flexible structure of the aggregation pipeline. Understanding which tool is appropriate for the task is crucial for writing performant and accurate queries. This article will first briefly address collection joining using **$lookup**, and then dive deep into the primary focus: using **$concat** within the **$project** stage to efficiently combine fields within documents, providing detailed examples for implementation.

Understanding Collection Joins with $lookup

Although the primary focus here is on intra-document field combination, it is essential to contextualize how $lookup facilitates inter-collection joining. The **$lookup** operator must be used as part of the aggregation pipeline and is specifically designed to perform a left outer join from a primary collection to a secondary collection. This powerful feature allows developers to effectively denormalize data on the fly by embedding matching documents from the "foreign" collection into an array field in the "local" collection's documents. This is invaluable when retrieving data from two related collections, such as linking user data to order history, and combining this disparate data into a single, cohesive result set for application use.

The core benefit of using **$lookup** is its adherence to left outer join semantics, meaning documents from the input collection are always included in the result, even if no corresponding matches exist in the joined collection. When structuring a **$lookup** stage, you must specify the foreign collection (`from`), the local field (`localField`), the field in the foreign collection to match against (`foreignField`), and the name of the new array field to output the results to (`as`). While this operator handles complex relationship mapping, for simple intra-document field merging, we utilize string manipulation operators within the pipeline, specifically the **$concat** expression.

The Role of the Aggregation Pipeline in Concatenation

All advanced data transformations, including field concatenation, rely on the Aggregation pipeline. The pipeline processes documents through multiple stages sequentially, allowing for filtering, grouping, reshaping, and precise transformation of the data. To concatenate strings and create a

new field, we utilize the $project stage. The **$project** stage is crucial because it allows us to define new fields based on existing data, or reshape the documents by selecting specific fields. This is where the actual concatenation logic resides, utilizing the $concat string expression operator.

The **$concat** operator accepts an array of strings (or expressions that resolve to strings) and outputs a single, combined string. The array dictates the precise sequence of concatenation. A critical consideration here is data integrity: if any of the input values are `null`, the entire **$concat** expression will return `null`, an important behavior to remember when dealing with potentially incomplete documents. After defining the new concatenated field in the **$project** stage, if the intention is to persist this new field back into the original collection, a subsequent stage like $merge is necessary, enabling us to update the collection with the newly calculated values.

Syntax for String Concatenation using $concat and $project

To concatenate strings from two fields into a single new field, the standard pattern involves passing the document through the **$project** stage. Within **$project**, you define the name of the desired new field (e.g., `newfield`). The value assigned to this field is an object containing the **$concat** operator. The **$concat** operator then takes an array where each element represents a segment of the final string. These segments can be references to existing fields (prefixed with `$`), or static string literals enclosed in quotation marks, such as separators or fixed prefixes.

The following syntax demonstrates the standard structure for this operation. Here, `$field1` and `$field2` represent the fields whose values are being combined. We insert a static string separator, `" - "`, between the two field values to improve readability. Finally, the $merge stage is included to write the resulting modified documents back into the original collection, specified here as `myCollection`, ensuring the data transformation is persisted beyond the query execution.

You can use the following syntax to concatenate strings from two fields into a new field in MongoDB:

db.myCollection.aggregate(} } },
{ $merge: "myCollection" }
])

This sequence efficiently concatenates the strings derived from `field1` and `field2`, inserting the custom separator, and storing the resulting combined string into the new field named `newfield`. By utilizing the **$merge** stage, the calculated data is automatically integrated into the existing `myCollection` documents, updating them in place. This particular example concatenates the strings from "field1" and "field2" into a new field titled "newfield" and adds the new field to the collection titled `myCollection`.

Practical Example Setup: Initializing the Data

To provide a concrete illustration of the concatenation functionality, we will establish a sample collection named `teams`. This collection holds basic information about sports teams, including the team name, their conference affiliation, and their current point total. Our objective is to combine the `team` name and the `conference` designation into a single, descriptive field. This combined field, which we will call `teamConf`, provides a unique and easily identifiable label for each record.

We will initialize the `teams` collection with six documents, representing teams from both the Western and Eastern conferences. Each document is inserted using the **insertOne** command, ensuring that the data structure is consistent across all entries, which is crucial for reliable aggregation pipeline processing. The following example shows how to use this syntax in practice with the collection `teams` with the following documents:

db.teams.insertOne({team: "Mavs", conference: "Western", points: 31})
db.teams.insertOne({team: "Spurs", conference: "Western", points: 22})
db.teams.insertOne({team: "Rockets", conference: "Western", points: 19})
db.teams.insertOne({team: "Celtics", conference: "Eastern", points: 26})
db.teams.insertOne({team: "Cavs", conference: "Eastern", points: 33})
db.teams.insertOne({team: "Nets", conference: "Eastern", points: 38})

This data setup provides a clear and repeatable foundation for our demonstration. We now have a collection of documents ready to be processed by the aggregation pipeline. The next steps will involve constructing the specific **$project** stage that executes the string combination logic, creating the desired composite field for each team record.

Example: Implementing Concatenation with a Separator

A fundamental requirement in string combination is often the inclusion of a separator--such as a dash, space, or underscore--to enhance readability and clarity in the resultant field. For our `teams` collection, we want to create the new field `teamConf` by combining the `team` name and the `conference`, using a dash (`-`) as the delimiter. This ensures the concatenated value, such as "Mavs - Western," remains easily decipherable and user-friendly.

We can use the following code to concatenate the strings from the "team" field and the "conference" field into a new field titled "teamConf" and add this field to the `teams` collection. The $project stage is configured to output the necessary fields, including the newly created `teamConf` field. Within the $concat array, we sequence the field references (`$team` and `$conference`) with the static string literal `" - "` placed exactly in the middle. Following the transformation, we apply the $merge stage, targeting the original `teams` collection, ensuring persistence of the new field.

The complete aggregation pipeline execution for concatenating the team and conference fields with a separator is defined below:

db.teams.aggregate(} } },
{ $merge: "teams" }
])

Here's what the updated collection now looks like after executing the aggregation. Notice that every document has a new field titled "teamConf" that contains the concatenation of the "team" and "conference" fields. For this particular example we chose to concatenate the two strings together using a dash as a separator.

{ _id: ObjectId("62013d8c4cb04b772fd7a90c"),
team: 'Mavs',
conference: 'Western',
points: 31,
teamConf: 'Mavs - Western' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90d"),
team: 'Spurs',
conference: 'Western',
points: 22,
teamConf: 'Spurs - Western' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90e"),
team: 'Rockets',
conference: 'Western',
points: 19,
teamConf: 'Rockets - Western' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90f"),
team: 'Celtics',
conference: 'Eastern',
points: 26,
teamConf: 'Celtics - Eastern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a910"),
team: 'Cavs',
conference: 'Eastern',
points: 33,
teamConf: 'Cavs - Eastern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a911"),
team: 'Nets',
conference: 'Eastern',

points: 38,
teamConf: 'Nets - Eastern' }

Concatenating Without Separators for Compact Identifiers

While incorporating a separator generally improves readability for human consumption, there are specific scenarios where concatenation must occur without any intermediate characters. This is often necessary when constructing unique database keys, generating URL slugs, or creating internal system identifiers where spaces or special characters are inappropriate. If we revisit the `teams` example and choose to concatenate the `team` and `conference` fields directly, the resulting `teamConf` field will merge the strings adjacently, such as "MavsWestern."

The adjustment required in the $project stage is straightforward: we simply remove the static string literal (`" - "`) from the array passed to the **$concat** operator. The input array now consists solely of the field references `"$team"` and `"$conference"` in the desired sequence. The following code shows how to do so:

db.teams.aggregate(} } },
{ $merge: "teams" }
])

Executing this revised aggregation updates the collection once more, demonstrating how flexibility allows for different output formats. This method is highly effective for creating compact, unique identifiers derived from multiple source fields. Here's what the updated collection would look like:

{ _id: ObjectId("62013d8c4cb04b772fd7a90c"),
team: 'Mavs',
conference: 'Western',
points: 31,
teamConf: 'MavsWestern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90d"),
team: 'Spurs',
conference: 'Western',
points: 22,
teamConf: 'SpursWestern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90e"),
team: 'Rockets',
conference: 'Western',
points: 19,

teamConf: 'RocketWestern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a90f"),
team: 'Celtics',
conference: 'Eastern',
points: 26,
teamConf: 'CelticsEastern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a910"),
team: 'Cavs',
conference: 'Eastern',
points: 33,
teamConf: 'CavsEastern' }
{ _id: ObjectId("62013d8c4cb04b772fd7a911"),
team: 'Nets',
conference: 'Eastern',
points: 38,
teamConf: 'NetsEastern' }

Important Data Type and Null Value Considerations

When working with string concatenation in MongoDB, developers must be mindful of potential pitfalls, particularly concerning data types and null values. The **$concat** operator strictly requires that all elements passed into its array resolve to strings. If a non-string field, such as a number or a date, is referenced without being explicitly converted to a string using operators like **$toString**, the aggregation will fail and return an error. Ensuring explicit type casting before the concatenation step is a critical best practice for robust query construction.

Furthermore, if any element within the array passed to **$concat** resolves to `null` or references a field that is missing entirely from a document, the entire **$concat** expression for that document will evaluate to `null`. This means the resulting concatenated field will either be missing or explicitly null, potentially leading to inconsistencies. To mitigate such null value issues, advanced aggregation techniques involve using the **$ifNull** or **$cond** operators prior to concatenation, allowing developers to substitute a safe default value, such as an empty string (`""`), if a source field is missing or null.

For those looking to perform more complex data joining operations across collections, detailed documentation on the $lookup operator is essential. Conversely, if your focus remains on advanced string manipulation, such as slicing, sub-string extraction, or case conversion, the comprehensive documentation for the $concat function and related string operators within the aggregation pipeline will provide the necessary details for robust data transformation.

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

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