# Learning to Concatenate Strings from Two Fields in MongoDB Aggregations

Authored by

**Mohammed looti**

October 31, 2025

One of the most common requirements in data transformation is combining data from multiple fields into a single, cohesive unit. In MongoDB, achieving this requires leveraging the powerful Aggregation Pipeline. This article provides an expert guide on how to efficiently concatenate strings from two different fields within a document and persist the result back into your existing data structure.

## Understanding the Core Syntax for String Concatenation

To successfully combine string values from two distinct fields and map the result to a new field, we utilize a sequence of specific aggregation stages. The fundamental approach involves the **$project** stage to structure the output and the **$concat** operator to perform the actual string joining operation. Finally, we use **$merge** to update the collection in place.

The following standard syntax illustrates how you can concatenate strings from two source fields, `field1` and `field2`, inserting a separator (a hyphen and space) between them, and storing the final result in a new field named `newfield`:

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

This powerful sequence of operations transforms the documents processed by the pipeline. The **$merge** stage ensures that the newly calculated field, `newfield`, is seamlessly added to the original documents within the target myCollection. It is critical to understand that **$concat** accepts an array of strings or field references, allowing for complex combinations, including static separators like " - ".

## Setting Up the Sample Data

To demonstrate this functionality in a real-world scenario, let us establish a sample collection named teams. This collection tracks various sports teams, their conferences, and their current point totals. This setup provides ideal source material for combining descriptive fields like the team name and its conference designation.

We will populate the teams collection with the following six sample 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})**

Our objective is to create a new field, `teamConf`, that combines the team name and its respective conference, providing a unique and descriptive identifier for each entry.

## Practical Example 1: Concatenating Strings with a Separator

In many database scenarios, when merging two distinct strings, it is essential to include a separator character--such as a hyphen, space, or comma--to maintain readability and clarity. This is particularly important when the resulting concatenated string is intended for human consumption or reporting purposes. We will apply this technique to combine the values from the `team` field and the `conference` field, using " - " as the delimiter.

The following aggregation command executes this concatenation, creates the new `teamConf` field, and updates the teams collection by merging the results back into the original documents:

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

Upon successful execution of the aggregation pipeline, every document in the collection now possesses the newly calculated field. The output below demonstrates the structure of the updated documents, clearly showing the `teamConf` field containing the combined string formatted as "Team - Conference":

**{ _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' }**

Observe how the **$concat** array successfully handled three distinct elements: the field reference `$team`, the static string literal `" - "`, and the field reference `$conference`. This demonstrates the flexibility of the operator in combining dynamic data with fixed formatting elements.

## Practical Example 2: Concatenating Strings Without a Separator

While separators enhance readability, there are scenarios--such as generating compact unique identifiers, keys, or slugs--where omitting any delimiter is preferred. Generating a tight, contiguous string often results in cleaner data when integrating with external systems or APIs that have strict formatting rules for identifiers.

If we choose to concatenate the `team` and `conference` fields without any intervening static characters, the structure of the **$concat** array simplifies considerably. We simply list the two field references in the desired order within the array, as shown in the code below:

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

Executing this modified aggregation updates the `teamConf` field across all documents. The resulting strings are merged directly against one another, eliminating the space and the dash that were present in the previous example.

The updated collection documents now reflect this change, showing compact concatenated strings

in the `teamConf` field:

**{ _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: 'RocketsWestern' }**
**{ _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' }**

## Key Considerations for Using $concat and $merge

When implementing data transformations that modify existing documents, understanding the implications of the stages used is vital for performance and data integrity.

The **$concat** operator is designed strictly for string manipulation. If any input field passed to

**$concat** contains a `null` value or is missing from a document, the entire concatenation operation for that document will result in `null`. Therefore, when working with potentially sparse data, it is often necessary to precede the **$project** stage with a **$ifNull** or a **$cond** operator to substitute missing values with empty strings (`""`) before concatenation occurs. This prevents unintended data loss during the aggregation process.

Furthermore, the choice of persistence stage is critical. We used **$merge**, which updates the input collection itself. If you prefer to avoid modifying the original data and instead want to create a new, transformed collection, you would use the **$out** stage instead of **$merge**. While **$merge** is excellent for transactional, incremental updates, **$out** is safer for large, full-data transformations.

**Note**: You can find the complete and authoritative documentation for the **$concat** aggregation function on the official MongoDB website.

## Additional Resources

To further enhance your skills in MongoDB data manipulation and aggregation techniques, consider exploring the following related tutorials which explain how to perform other common operations:

Tutorial on calculating differences between dates in MongoDB.
Guide to using the $group stage for data summarization.
Detailed explanation of the $lookup operator for joining collections.