How do I merge cells with the same values in VBA?

Authored by

stats writer

November 19, 2025

Automating repetitive tasks within Microsoft Excel is a core function of Visual Basic for Applications (VBA). One common requirement in data presentation is the need to merge adjacent cells that contain identical values, transforming raw data into a clean, summarized report. Manually performing this action across hundreds or thousands of rows is inefficient and prone to error. This powerful technique leverages a macro that systematically iterates through a specified Range, compares the value of the current cell to the one immediately below it, and executes a merge operation if a match is found. This script provides an essential tool for data professionals looking to streamline the creation of summary tables or consolidate related information quickly and accurately.

The core logic hinges on defining the execution area, employing a conditional loop structure to evaluate cell contents, and then applying the Merge method. Since the merging process alters the structure of the data set, we utilize a special loop control mechanism, often the **GoTo** statement, to ensure that the script re-evaluates the modified range accurately after each successful merge operation. This prevents the script from skipping rows and guarantees that all consecutive, identical values are correctly consolidated into single, unified cells. The result is a dramatically cleaner and more readable spreadsheet, ideal for reporting and visualization purposes.

This tutorial will guide you step-by-step through the construction of this specific VBA routine. We will detail the necessary syntax, explain the function of key commands such as **Application.DisplayAlerts**, and demonstrate how to handle the looping structure to achieve a robust and reliable merging solution. Mastering this technique allows for high levels of efficiency when dealing with large datasets requiring hierarchical or grouped presentation styles.

Understanding the VBA Method for Merging

The procedure for merging cells based on identical values requires a custom macro written in the VBA environment. This script must manage three critical tasks: initialization (setting parameters and disabling alerts), iteration (looping through the defined area), and execution (performing the merge and resetting the loop). We employ the **For Each** loop construct to ensure every cell within the chosen data block is evaluated against its subsequent counterpart. This ensures comprehensive coverage of the entire dataset specified by the user.

The key challenge when merging cells dynamically is that the act of merging two cells into one effectively shifts the structure of the remaining rows, potentially causing standard loops (like simple **For...Next** loops iterating by row number) to skip the newly shifted row immediately following the merge. To counteract this issue, we utilize the **GoTo** statement. When a successful merge occurs, the **GoTo MergeSame** command forces the code to jump back to the beginning of the merging loop structure. This critical step ensures that the script immediately re-evaluates the entire range from the top, capturing any remaining identical adjacent cells that might have been shifted into

position by the preceding merge operation. This recursive approach guarantees a complete and accurate merge across the entire defined Range.

Below is the specific VBA syntax required to successfully execute this process. Note the use of specialized functions and properties, such as Application.DisplayAlerts, which prevents disruptive confirmation dialog boxes from appearing during the merging process, thus allowing the macro to run smoothly without interruption.

```
Sub MergeSameCells()

'turn off display alerts while merging
Application.DisplayAlerts = False

'specify range of cells for merging
Set myRange = Range("A1:C13")

'merge all same cells in range
MergeSame:
For Each cell In myRange
If cell.Value = cell.Offset(1, 0).Value And Not IsEmpty(cell) Then
Range(cell, cell.Offset(1, 0)).Merge
cell.VerticalAlignment = xlCenter
GoTo MergeSame
End If
Next

'turn display alerts back on
Application.DisplayAlerts = True

End Sub
```

In this initial setup, the Range targeted for modification is explicitly set to **A1:C13**. This configuration is easily adaptable; users only need to update the string argument within the **Range()** function to cover any contiguous block of cells they wish to process. This flexibility ensures the macro can be reused across various spreadsheets and data structures without requiring extensive modifications to the core logic.

Line-by-Line Breakdown of the VBA Script

To fully grasp the mechanism of this powerful script, a detailed look at each key instruction is necessary. Understanding these components is vital for troubleshooting and customizing the

macro for different data requirements.

`Sub MergeSameCells()` / `End Sub`: These lines define the beginning and end of the subroutine, giving the macro its executable name.

`Application.DisplayAlerts = False`: Merging cells typically generates a warning message from Excel, notifying the user that merging only preserves the top-left value. Setting this property to **False** suppresses all such warnings, allowing the script to execute seamlessly. It is critical to set this back to **True** at the end of the script to restore standard Excel functionality.

`Set myRange = Range("A1:C13")`: This line defines a variable, **myRange**, as the specific area in the worksheet where the merging operation will take place. This is the primary area that must be adjusted for different data inputs.

`MergeSame:`: This is a label used as the target for the **GoTo** command. Whenever the code encounters a successful merge, it jumps back to this label to restart the loop.

`For Each cell In myRange`: This initiates the iterative process, ensuring that the instructions within the loop are applied sequentially to every single cell object within the defined **myRange**.

`If cell.Value = cell.Offset(1, 0).Value And Not IsEmpty(cell) Then`: This is the core conditional statement. It checks if the value of the current **cell** is exactly equal to the value of the cell one row below it (achieved using the Offset property, 1 row down, 0 columns across). The additional check, **And Not IsEmpty(cell)**, ensures that blank cells are not inadvertently processed or merged, preventing issues with potentially empty ranges.

`Range(cell, cell.Offset(1, 0)).Merge`: If the condition is met (values are identical), this command executes the merge operation, combining the current cell with the cell immediately below it.

`cell.VerticalAlignment = xlCenter`: This important formatting instruction ensures that once the cells are merged, the text content is visually pleasingly centered within the vertically extended cell, enhancing readability.

`GoTo MergeSame`: This crucial command dictates the flow control. Upon a successful merge, the loop breaks, and execution returns to the **MergeSame:** label. This restarts the iteration, guaranteeing that cells shifted up by the merge operation are not overlooked in the subsequent check.

Setting Up the Environment and Data

Before executing the macro, ensure you have correctly accessed the VBA Editor (Alt + F11 in

Excel) and inserted a new module (Insert > Module). The code must be pasted into this module. Furthermore, it is essential that the data you intend to merge is sorted correctly. The macro only merges adjacent cells with the same value; therefore, if identical values are separated by different values, they will not be merged. Proper sorting, usually by the columns you intend to merge, is a fundamental prerequisite for this script to function as intended.

The structure of the data needs careful consideration. While this macro is designed to merge vertically (combining a cell with the cell directly beneath it), its effectiveness is maximized when used on columns that define categories or groupings, such as names, departments, or dates. Applying this technique to numerical data, such as scores or quantities, is less common unless those numbers represent fixed categories. Always perform a backup of your worksheet before running any powerful macro that alters cell structure, as merging is an irreversible operation.

For this specific example, the data structure focuses on organizational categories that repeat, making it an ideal candidate for vertical consolidation. We are aiming to visually group related rows under a single heading cell, enhancing the overall clarity of the displayed information.

Practical Application: Merging Dataset Categories

The following example illustrates how this technique is applied to a practical dataset. Suppose we have compiled data detailing points scored by basketball players, organized by Conference and Team. We want to clean up the presentation so that repeating Conference and Team names are merged into single cells, clearly delineating the different groups of players.

The original dataset appears as follows, exhibiting repetitive entries in the first two columns:

	A	B	C	D	E	F
1	**Conference**	**Team**	**Points**			
2	Western	Mavs	22			
3	Western	Mavs	14			
4	Western	Rockets	11			
5	Western	Spurs	19			
6	Western	Spurs	14			
7	Western	Spurs	29			
8	Eastern	Celtics	30			
9	Eastern	Celtics	35			
10	Eastern	Celtics	36			
11	Eastern	Hornets	31			
12	Eastern	Hornets	18			
13	Eastern	Hornets	15			
14						
15						
16						
17						
18						
19						

Our objective is to consolidate the identical values in consecutive rows within the designated Range (A1:C13 in this instance). This transforms the repetitive data entries into a hierarchical summary view, making it immediately apparent which players belong to which teams and conferences.

We will utilize the identical macro structure defined previously, ensuring the Range variable **myRange** is set correctly to cover the full extent of the data columns we wish to process.

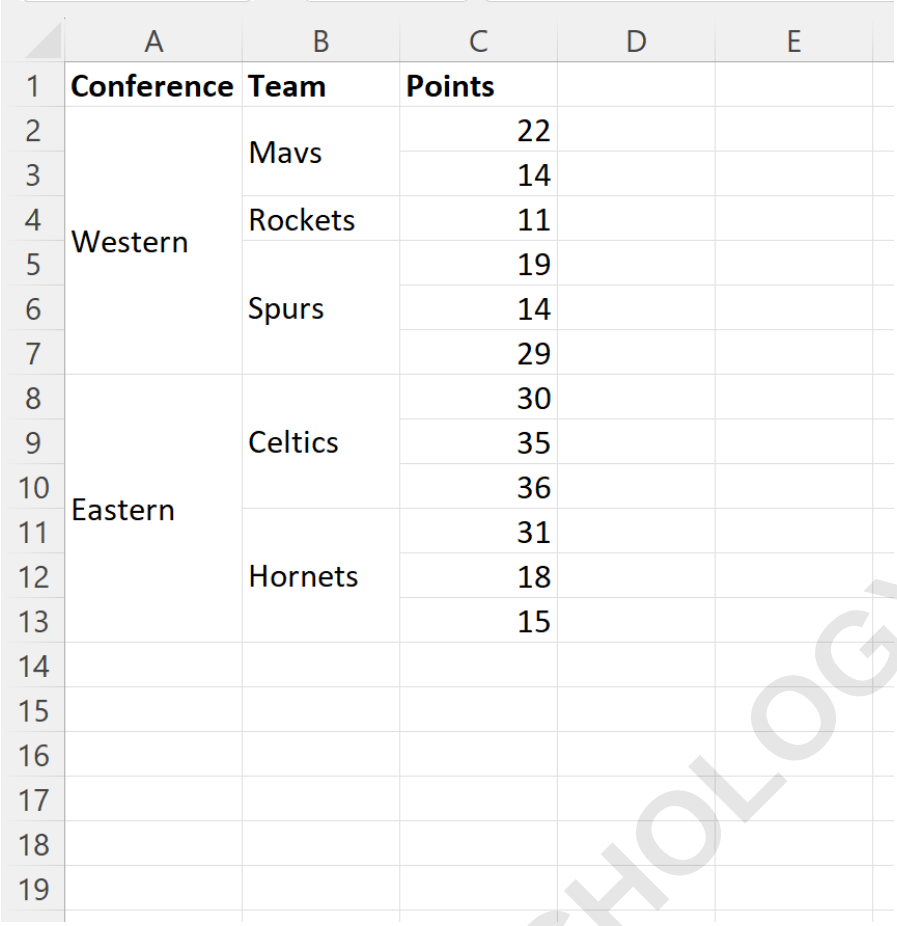
```
Sub MergeSameCells()

'turn off display alerts while merging
Application.DisplayAlerts = False

'specify range of cells for merging
Set myRange = Range("A1:C13")

'merge all same cells in range
MergeSame:
For Each cell In myRange
If cell.Value = cell.Offset(1, 0).Value And Not IsEmpty(cell) Then
```

```
Range(cell, cell.Offset(1, 0)).Merge
cell.VerticalAlignment = xlCenter
GoTo MergeSame
End If
Next

'turn display alerts back on
Application.DisplayAlerts = True

End Sub
```

Reviewing the Merged Output

Upon successfully running the **MergeSameCells** subroutine, the worksheet immediately reflects the structural changes dictated by the VBA code. The result is a much more consolidated table where the categorical information is visually distinct and non-redundant.

The output following the macro execution demonstrates the effectiveness of the recursive looping structure and the merging command:



	A	B	C	D	E
1	**Conference**	**Team**	**Points**		
2	Western	Mavs	22		
3			14		
4		Rockets	11		
5		Spurs	19		
6			14		
7			29		
8	Eastern	Celtics	30		
9			35		
10			36		
11		Hornets	31		
12			18		
13			15		
14					
15					
16					
17					
18					
19					

As clearly demonstrated by the image, the cells containing the same **Conference** names (e.g., Eastern) and subsequent **Team** names have been successfully merged. This transformation is highly beneficial for large reports, as it minimizes visual clutter and focuses the reader's attention on the distinct data groups. The data is now grouped logically, providing a clear boundary for each unit of analysis.

Key Considerations for Merging Cells

While merging cells enhances visual presentation, users must be aware of potential drawbacks, particularly concerning subsequent data manipulation. Merged cells often interfere with standard Excel functions like filtering, sorting, and pivot table creation, which typically expect data to reside in single, discrete cells. Therefore, this merging technique is best employed as the final step in data preparation, specifically for reporting or printing purposes, after all analytical tasks have been completed.

A crucial element in the script is the line **cell.VerticalAlignment = xlCenter**. This statement is essential for aesthetic purposes. When two or more cells are merged vertically, the text by default remains aligned at the top of the newly formed larger cell. By applying **xlCenter**, we ensure that

the content is centered vertically, which looks professional and clearly identifies the merged cell as the header or identifier for the associated subsequent rows of data. Without this alignment command, the resulting merged cells can appear unbalanced.

Furthermore, the use of the Offset property in the conditional check (**cell.Offset(1, 0)**) is instrumental. It specifies that the comparison must only occur with the cell immediately below the current cell (one row down, zero columns right). If the goal were to merge identical values horizontally, the arguments for the **Offset** function would need to be adjusted accordingly (e.g., 0 rows down, 1 column right), demonstrating the script's adaptability based on the direction of merging required.

Advanced Customization and Error Handling

For more complex datasets, this macro can be easily modified to handle multiple columns simultaneously or incorporate more robust error handling. For instance, if you only wanted to merge cells in Column A and Column C, you would adjust the initial **myRange** definition and potentially run separate looping structures for each column, ensuring that the merging logic remains focused only on adjacent cells within that specific column boundary.

To enhance the robustness of the script, especially in corporate environments where worksheets might be locked or protected, additional error handling mechanisms should be considered. Utilizing VBA's **On Error GoTo** structure allows the macro to gracefully handle unexpected situations, such as attempting to merge cells in a read-only worksheet, preventing the script from crashing abruptly. A simple error block could inform the user of the issue and ensure that **Application.DisplayAlerts** is switched back to **True** even if the main merging process fails.

Possible modifications and enhancements include:

Dynamic Range Selection: Modifying the script to automatically determine the last used row and column, making the macro independent of predefined boundaries like **A1:C13**.

Column Specific Merging: Introducing an argument into the subroutine allowing the user to specify which column(s) should be processed, rather than forcing the script to run across all columns in the defined range.

Visual Feedback: Temporarily changing the interior color of the cells being processed, offering visual confirmation that the macro is actively working, especially when dealing with extremely large datasets that take time to execute.

By implementing these adjustments, this foundational VBA script evolves from a basic utility into a highly versatile and professional tool for comprehensive data management and reporting within Excel.