How to Use Intersect in VBA (With Examples)

Authored by

stats writer

November 18, 2025

Introduction to the VBA Intersect Method

The Intersect Method is a highly valuable function within VBA, primarily utilized to determine the common area shared by two or more defined ranges on an Excel worksheet. This method returns a new Range object, which precisely represents the intersection of the input ranges. This capability is fundamental for automating complex calculations, setting dynamic conditional formatting, or implementing advanced event handling where code execution depends on where a user is interacting within the **spreadsheet**.

The Intersect function belongs to the Application object, meaning it operates globally within the Excel environment. Its robust design allows it to handle various scenarios, whether the intersection results in a single cell, a multi-cell rectangular region, or even no overlap at all. By isolating overlapping regions, developers can focus their code execution or data extraction efforts on only the most relevant parts of a **dataset**, thereby increasing efficiency and minimizing potential errors associated with manually defining boundaries.

Defining the Custom FindIntersect Function

While the Intersect Method is typically used in **VBA Subroutines**, wrapping it within a User Defined Function (UDF) allows end-users to leverage its geometric power directly within the Excel spreadsheet interface. This UDF acts as a bridge, enabling the function to accept range references as standard formula arguments, making the intersection logic immediately accessible for routine data analysis without needing to run a macro.

The code block below illustrates the implementation of a simple UDF named **FindIntersect**. This function accepts two range parameters (`range1` and `range2`) and uses the Application object to call the intrinsic Intersect method. The result of this intersection calculation is then assigned back to the Function name, allowing it to be returned to the worksheet cell.

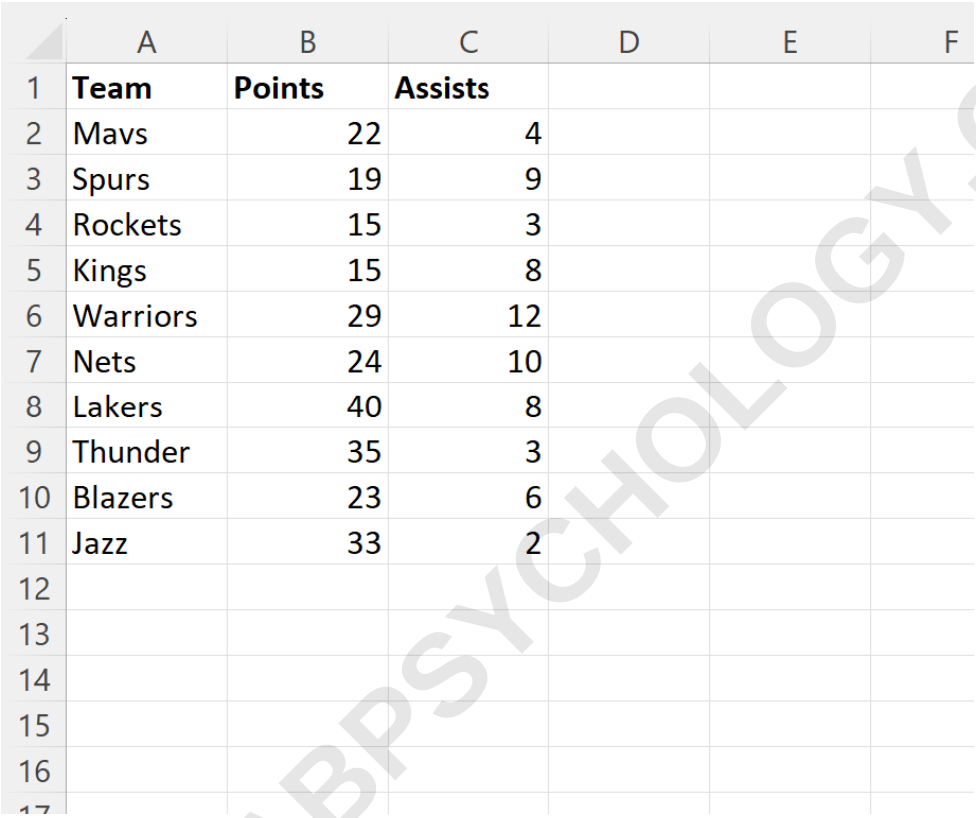
```
Function FindIntersect(range1 As Range, range2 As Range)
Set FindIntersect = Application.Intersect(range1, range2)
End Function
```

When this **FindIntersect** UDF is used in a worksheet cell, it evaluates the content of the intersecting Range. If the intersection is a single cell, the function returns that cell's value. If the intersection is a multi-cell Range, it generally returns the value of the top-left cell, unless entered as an **array formula**, in which case it returns all values within the overlap.

Prerequisites: Understanding the Sample Dataset

For clarity, all subsequent examples will reference the same sample dataset, which contains structured information about various items across multiple columns. This consistent base allows us to clearly visualize how different range selections interact and how the **VBA Intersect Method** resolves these overlaps.

The following image displays the data we will be using, spanning columns A through C and multiple rows. We will specifically define ranges based on this layout to test both single-cell and multi-cell intersection scenarios.



	A	B	C	D	E	F
1	**Team**	**Points**	**Assists**			
2	Mavs	22	4			
3	Spurs	19	9			
4	Rockets	15	3			
5	Kings	15	8			
6	Warriors	29	12			
7	Nets	24	10			
8	Lakers	40	8			
9	Thunder	35	3			
10	Blazers	23	6			
11	Jazz	33	2			
12						
13						
14						
15						
16						

In our forthcoming examples, we will precisely define the input ranges based on cell coordinates (e.g., **A2:C2** or **A1:B10**). This detail ensures that the resulting intersection is mathematically predictable, allowing us to confirm the correct execution of the **FindIntersect** custom function.

Example 1: Isolating a Single Cell Intersection

A common requirement in data analysis is finding the specific value at the point where a particular row and column converge. The Intersect Method excels at this task. In this example, we aim to isolate a single cell based on the overlap of one horizontal range and one vertical range.

We define the two ranges to be compared against the sample dataset:

Range 1: **A2:C2** (The entire second row of data shown).
Range 2: **A1:A11** (The entire first column, encompassing all item names).

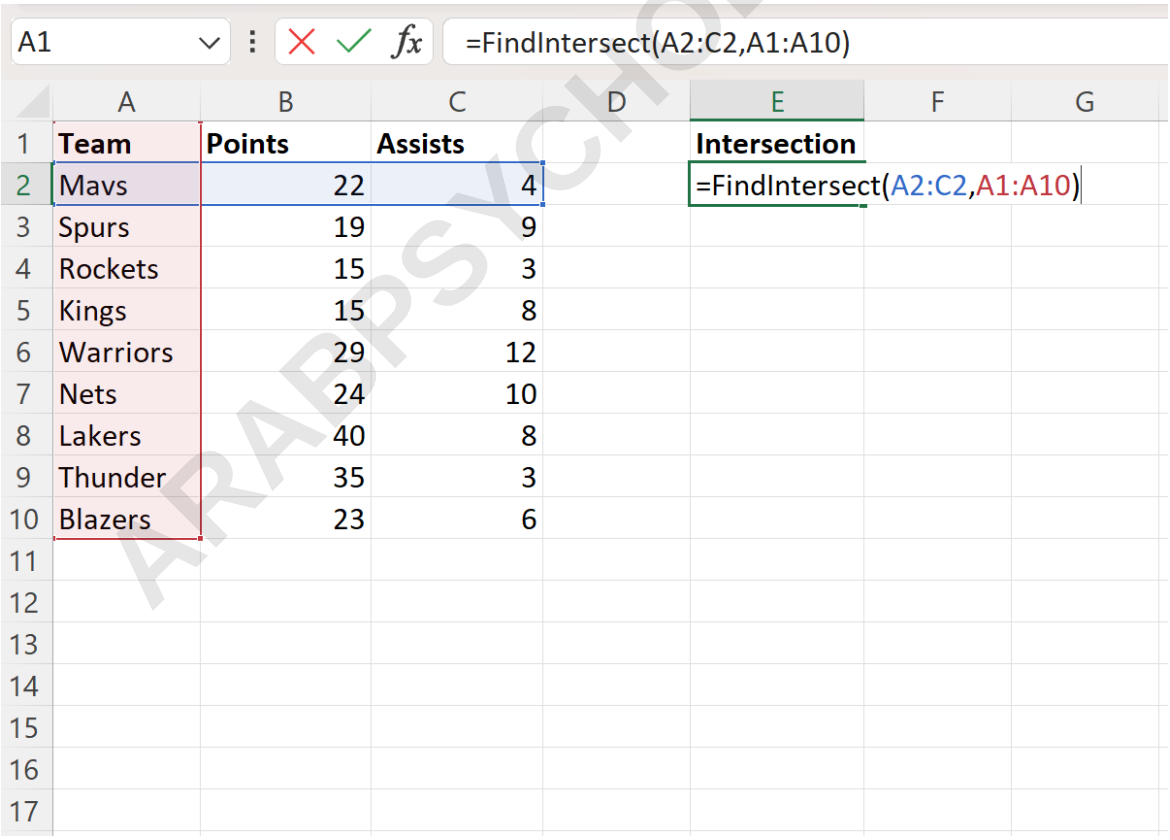
The only cell common to both ranges is **A2**. When we apply the **FindIntersect** function, we expect it to return the value held within that singular intersecting cell, thereby confirming the function's ability to pinpoint specific data points defined by **spatial criteria**.

Implementation and Result of Single Cell Intersection

The same UDF defined earlier is used. We enter the formula directly into a spare cell (e.g., E2) on the spreadsheet, providing the range references as arguments.

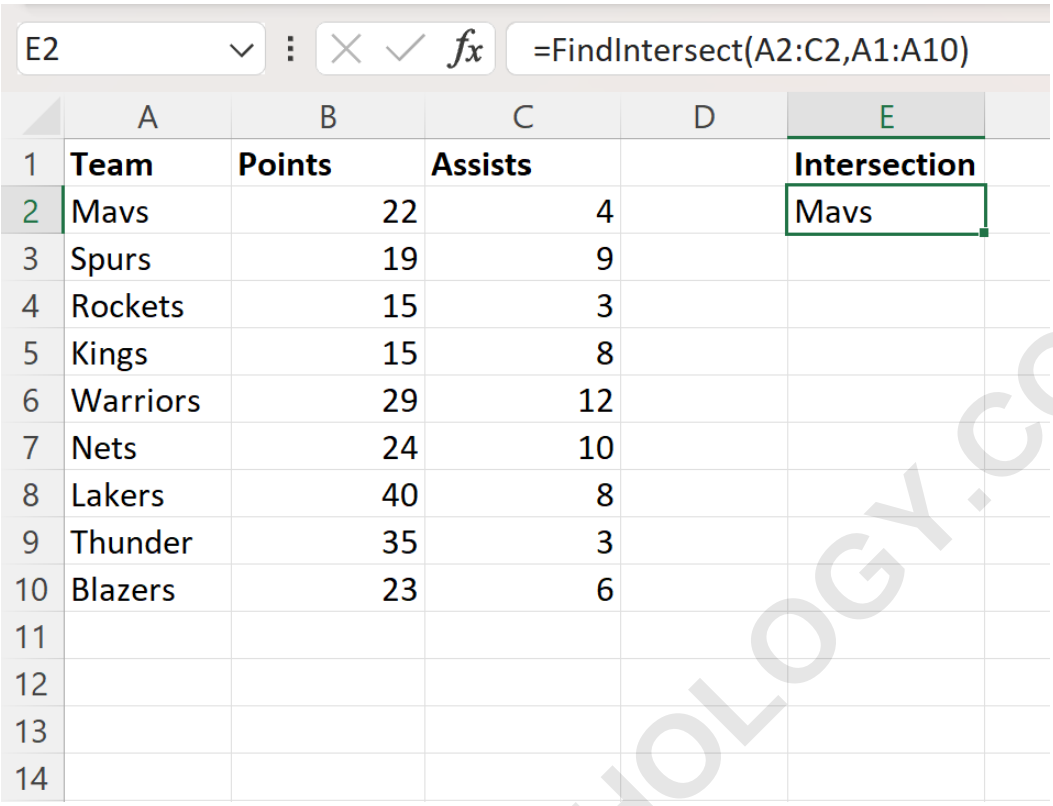
```
Function FindIntersect(range1 As Range, range2 As Range)
Set FindIntersect = Application.Intersect(range1, range2)
End Function
```

The input formula looks like this:



As anticipated, the formula successfully returns the value **Mavs**, which is the content of cell A2. This outcome verifies that the **VBA** Intersect Method correctly calculated the intersection between

the horizontal range **A2:C2** and the vertical range **A1:A11**.



	A	B	C	D	E
1	**Team**	**Points**	**Assists**		**Intersection**
2	Mavs	22	4		Mavs
3	Spurs	19	9		
4	Rockets	15	3		
5	Kings	15	8		
6	Warriors	29	12		
7	Nets	24	10		
8	Lakers	40	8		
9	Thunder	35	3		
10	Blazers	23	6		
11					
12					
13					
14					

Example 2: Returning a Multi-Cell Range Intersection

Beyond single-cell identification, the Intersect Method is exceptionally useful for dynamically selecting a contiguous block of data. This occurs when both input ranges are large and share a rectangular area of overlap. This second example demonstrates how to extract multiple cells simultaneously.

We define two overlapping ranges against our dataset:

Range 1: **A1:C3** (Top three rows, first three columns).
Range 2: **A1:B10** (Top ten rows, first two columns).

The intersection of these two areas is geometrically defined by the smallest common boundaries: the cells extending from column A to column B, and from row 1 to row 3. The expected result is the entire Range **A1:B3**.

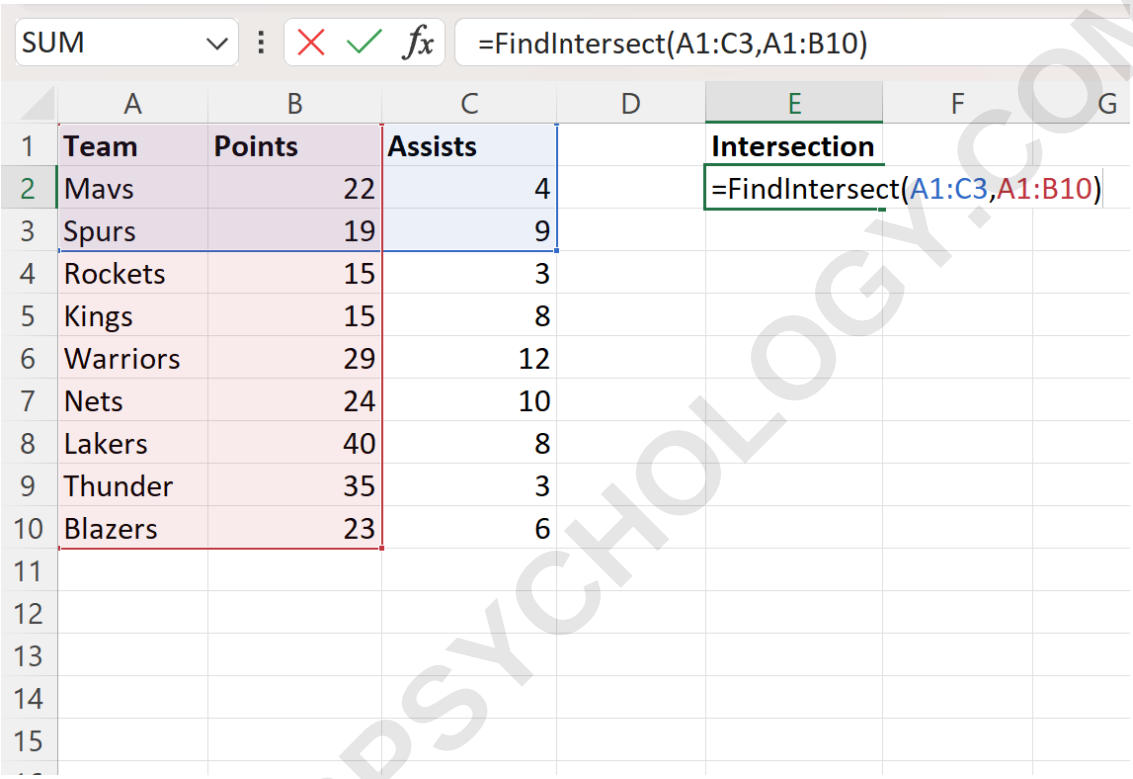
Implementation and Visualization of Multi-Cell Intersection

We utilize the identical **FindIntersect** Function defined earlier, as the underlying Application object

handles the complexity of the range calculation regardless of the size of the overlap.

```
Function FindIntersect(range1 As Range, range2 As Range)
Set FindIntersect = Application.Intersect(range1, range2)
End Function
```

When entered into the spreadsheet:



	Team	Points	Assists
2	Mavs	22	4
3	Spurs	19	9
4	Rockets	15	3
5	Kings	15	8
6	Warriors	29	12
7	Nets	24	10
8	Lakers	40	8
9	Thunder	35	3
10	Blazers	23	6

The resulting output showcases the extracted data block. Because modern Excel versions support dynamic arrays, the UDF is capable of spilling the results of the entire intersection Range (**A1:B3**) into neighboring cells. This confirms that the formula successfully identified the full rectangular area that intersects the ranges **A1:C3** and **A1:B10**.

E2 =FindIntersect(A1:C3,A1:B10)

	A	B	C	D	E	F
1	**Team**	**Points**	**Assists**		**Intersection**	
2	Mavs	22	4		Team	Points
3	Spurs	19	9		Mavs	22
4	Rockets	15	3		Spurs	19
5	Kings	15	8			
6	Warriors	29	12			
7	Nets	24	10			
8	Lakers	40	8			
9	Thunder	35	3			
10	Blazers	23	6			
11						
12						
13						
14						
15						

Advanced Considerations and Best Practices

When working with the Intersect Method in more complex **VBA** code (i.e., not just a UDF), it is essential to manage the case where no intersection exists. If the input ranges do not overlap, Intersect returns the value **Nothing**. Failure to check for this condition can lead to runtime errors when attempting to manipulate the properties of the resulting Range object.

The correct syntax for checking for no overlap in a **VBA Subroutine** is: `If Application.Intersect(RangeA, RangeB) Is Nothing Then...`. Additionally, while our examples used only two ranges, the Intersect Method can handle multiple ranges simultaneously, allowing up to 30 range arguments, finding the common area across all specified inputs. This makes it an incredibly versatile tool for filtering and validation tasks across large **VBA projects**.