Learning to Combine Data: Using CONCAT and QUERY Functions in Google Sheets

Authored by

Mohammed looti

October 30, 2025

The Crucial Need for Dynamic Concatenation in Google Sheets

In the realm of data management using Google Sheets, users frequently encounter the necessity of transforming raw, tabular data into more readable or structured formats. A primary method for achieving this transformation is **concatenation**, which involves joining the textual content of several columns into a single, comprehensive cell for every record. This technique is indispensable for generating unique keys, assembling standardized addresses, or creating simplified reports where related information must be presented cohesively. While Sheets provides straightforward tools for string manipulation, such as the CONCATENATE function (or its modern, shorter form, **CONCAT**), and features the incredibly powerful QUERY function for advanced filtering and selection, combining these two capabilities seamlessly presents a significant hurdle.

The challenge arises because a conventional application of **CONCAT** cannot be directly embedded within a **QUERY** formula to perform row-wise aggregation dynamically. The core purpose of the **QUERY** function is to return a structured two-dimensional array of data, adhering to SQL-like logic for selection and filtering. It is not inherently designed to perform complex string aggregation across multiple columns simultaneously for each resulting row. Consequently, users seeking to filter data and then combine specific fields often find themselves searching for an elegant, single-cell formula that effectively marries the selection prowess of **QUERY** with the string-joining utility of **CONCAT**.

Fortunately, a sophisticated, multi-layered formula exists that leverages several advanced functions in concert to overcome this fundamental limitation. This powerful workaround enables you to execute complex data filtering based on specific criteria and subsequently concatenate the data from the selected columns into a single, clean output cell for every row returned by your query. This comprehensive guide will meticulously deconstruct this advanced technique, explaining the role of each component and providing clear, practical examples to demonstrate its implementation.

Understanding the Structural Conflict Between QUERY and CONCAT

The incompatibility between direct use of **CONCAT** and the QUERY function stems from their fundamentally different operational models. The **QUERY** function is designed to interpret data based on SQL standards, resulting in a predictable two-dimensional table structure, or an array. It is optimized for column-level operations--selecting, grouping, and aggregating vertically--but it lacks the built-in capability to horizontally "flatten" the values of a selected row into a single string.

Conversely, the CONCATENATE family of functions is purely dedicated to string manipulation, typically requiring specific cell references or a defined range of cells to operate on. While one could manually apply **CONCATENATE** to the output of a query row by row, the primary objective in

advanced data analysis is to perform this operation dynamically across an entire filtered dataset. This necessity mandates the use of an ARRAYFORMULA, which presents complexity when the output array structure needs drastic re-shaping and manipulation.

The solution involves an ingenious combination of functions, including two instances of TRANSPOSE, the **QUERY** function used in an unconventional manner, the text cleanup functions TRIM and SUBSTITUTE, all expertly nested within an **ARRAYFORMULA**. This multi-stage process first reorients the data, forces the concatenation using a special **QUERY** syntax, cleans the resulting string, and finally returns the data to the correct orientation, bridging the functional gap between filtering and string joining.

Deconstructing the Advanced Concatenation Formula

To successfully combine the results of a QUERY operation into a single concatenated cell for each row, we rely on the following powerful and complex ARRAYFORMULA structure:

```
=ARRAYFORMULA(
SUBSTITUTE(
trim(transpose(query(transpose(your_query),,COLUMNS(your_query)))),"  ","_")
)
```

This formula is a carefully choreographed sequence of operations designed to manipulate the array structure until row-wise concatenation is possible. It might seem daunting initially, but understanding the precise function of each nested element reveals its elegance and power:

ARRAYFORMULA: This is the essential outer layer, allowing the formula to process the entire array returned by the initial query. It ensures that the operation expands automatically across all relevant rows, eliminating the need for manual dragging and copying.

SUBSTITUTE (Outer): This function performs the final crucial cleanup. It replaces the default space character--which is implicitly used by the internal **QUERY** operation to join values--with your desired **separator**, such as an underscore (_), a comma, or a hyphen.

TRIM: Nested just inside the substitution mechanism, **TRIM** cleans up the concatenated string. It removes any extraneous leading or trailing whitespace and reduces multiple internal spaces down to single spaces. This is vital because the internal array operations can sometimes introduce unintended spacing.

TRANSPOSE (Inner): This is the initial reorientation step. It flips the rows and columns of the output generated by `your_query`. If the query result is an N-row by M-column table, this inner transpose converts it into an M-row by N-column table. This switch is necessary to set up the next

step--using **QUERY** for concatenation.

QUERY (Inner, The Concatenator): This is the most inventive component. By using `QUERY(transposed_data,,COLUMNS(your_query))`, where the select clause is omitted (indicated by the double comma `,,`), and the third argument (the header count) is specified, the **QUERY** function forces the concatenation of all rows in the transposed data. Since these transposed rows represent the original columns, this trick successfully achieves row-wise concatenation from the perspective of the source data, joining elements with a space by default.

COLUMNS(your_query): This function dynamically calculates the number of columns in the result set of the primary query. This count is then passed as the header argument to the inner **QUERY**, which, as mentioned, is the parameter that triggers the concatenation behavior.

TRANSPOSE (Outer): The final step is to flip the data back. After the inner **QUERY** performs the concatenation, this outer **TRANSPOSE** converts the resulting concatenated column back into the desired vertical orientation, aligning the final, single-string output with the original filtered rows.

The string `"_"` at the end of the formula represents the delimiter you wish to insert between the concatenated fields. This placeholder can be easily modified to any string or character that suits your reporting needs.

Practical Implementation: A Step-by-Step Example

To illustrate the power and complexity of this formula, let us consider a concrete scenario. Assume we have a dataset in Google Sheets spanning columns A through C, containing the First Name, Middle Name, and Last Name, respectively, as shown below:

	A	B	C	D
1	**First Name**	**Middle Name**	**Last Name**	
2	Andy	Ike	Smith	
3	Bob	Doug	Miller	
4	Andy	Jake	Johnson	
5	Andy	Alex	Anderson	
6	Eric	Ford	Hendrick	
7	Frank	Arnold	Harris	
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

Our specific goal is twofold: first, we must filter this data to include only records where the first name (Column A) contains the substring "Andy"; second, we must combine the First Name, Middle Name, and Last Name into a single, delimited string using an underscore (_) as the separator.

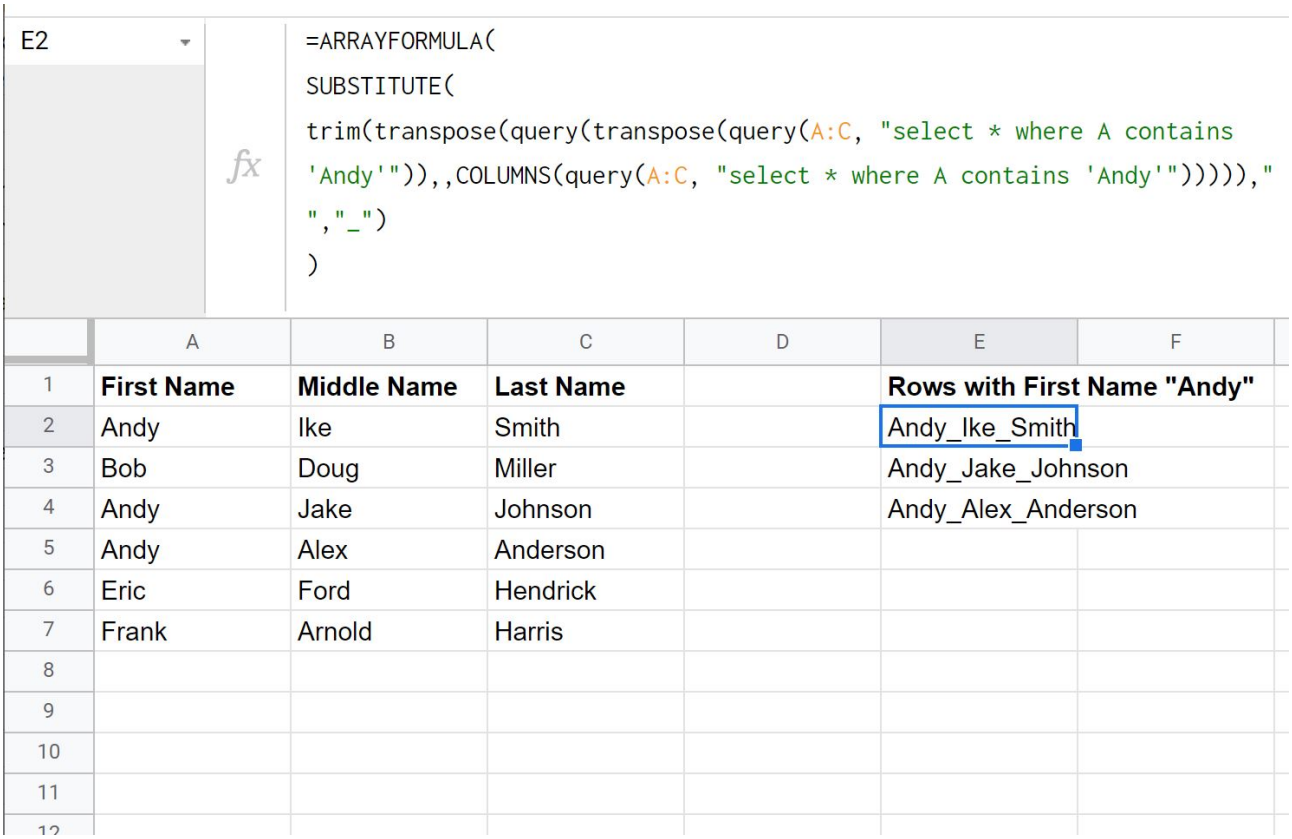
The specific formula required to execute this task in a single cell is constructed as follows:

=ARRAYFORMULA(
SUBSTITUTE(
trim(transpose(query(transpose(query(A:C, "select * where A contains 'Andy'")),,COLUMNS(query(A:C, "select * where A contains 'Andy'")))))," ","_")
)

In this application, the placeholder `your_query` is explicitly replaced by the expression `query(A:C, "select * where A contains 'Andy'")`. This inner **QUERY** is responsible for isolating the relevant subset of the data from the range `A:C`. The remaining, complex structure then takes this filtered subset, performs the necessary array manipulation via transposition, forces the row-wise aggregation using the internal query trick, cleans the resulting string with **TRIM**, and finally replaces the implicit joining spaces with the required underscore separator via the outer **SUBSTITUTE** function.

Analyzing the Results and Customizing Separators

When the advanced formula is applied to the example dataset, the output clearly demonstrates that both the filtering and the concatenation objectives have been met simultaneously, as shown in the screenshot below:



	A	B	C	D	E	F
1	**First Name**	**Middle Name**	**Last Name**		**Rows with First Name "Andy"**	
2	Andy	Ike	Smith		Andy_Ike_Smith	
3	Bob	Doug	Miller		Andy_Jake_Johnson	
4	Andy	Jake	Johnson		Andy_Alex_Anderson	
5	Andy	Alex	Anderson			
6	Eric	Ford	Hendrick			
7	Frank	Arnold	Harris			
8						
9						
10						
11						

The visual result confirms the success of the two main operations:

The formula accurately filters the original data, retaining only those rows where the First Name column (A) contains the specified search term, 'Andy'.

For every returned row, the values from the three name columns are seamlessly concatenated into a single cell. Critically, the formula correctly implements the specified separator--the underscore (_)--between each component of the name, resulting in a cohesive string format.

One of the key advantages of this method is its built-in flexibility regarding the separator. Should your requirements change, you can instantly modify the joining character or string by simply adjusting the final argument of the outermost SUBSTITUTE function.

For instance, if a standard space is preferred over an underscore to create a natural full name presentation, you would change `"_"` to `" "`. The following image illustrates the output when this

simple modification is made:

E2

```
=ARRAYFORMULA(
SUBSTITUTE(
trim(transpose(query(transpose(query(A:C, "select * where A contains 'Andy'")),,COLUMNS(query(A:C, "select * where A contains 'Andy'"))))),"  "," ")
)
```

	A	B	C	D	E	F
1	**First Name**	**Middle Name**	**Last Name**		**Rows with First Name "Andy"**	
2	Andy	Ike	Smith		Andy Ike Smith	
3	Bob	Doug	Miller		Andy Jake Johnson	
4	Andy	Jake	Johnson		Andy Alex Anderson	
5	Andy	Alex	Anderson			
6	Eric	Ford	Hendrick			
7	Frank	Arnold	Harris			
8						
9						
10						
11						
12						

In this revised output, the filtering criteria remain identical, but the first, middle, and last names are now gracefully joined by a space. This adaptability makes the formula exceptionally versatile for different reporting and data presentation requirements.

Key Takeaways and Best Practices for Implementation

The advanced ARRAYFORMULA detailed here represents a powerful and reliable solution to the challenge of performing row-wise concatenation immediately after filtering data using a **QUERY**. By grasping the specific roles of array-manipulation functions like TRANSPOSE, text-cleaning tools like TRIM, and string-replacement functions like SUBSTITUTE, users gain access to sophisticated data manipulation capabilities within Google Sheets.

To ensure successful and efficient deployment of this complex formula, consider adhering to the following best practices:

Optimize the Inner Query: Because the complex array operations are performed on the result of the inner query, keep the query itself as efficient as possible. Define the range precisely (e.g., `A:C` instead of `A:Z`) and use selective criteria to minimize the resulting data size.

Define the Delimiter Clearly: The choice of separator must be functional for the end goal. For human readability, a space or hyphen works best; for technical identifiers, underscores or pipes are often preferred.

Validate with Test Data: Given the nested nature of the functions, always test the formula against a small, known dataset to confirm that the filtration criteria are met and the concatenation is executed correctly before applying it to production data.

Monitor Performance: While this technique is robust, using multiple nested array formulas on extremely large datasets (hundreds of thousands of rows) may introduce latency. Be mindful of performance implications on heavy sheets.

Mastering this advanced technique is invaluable for generating highly customized reports, preparing aggregated fields for external analysis, and presenting complex information in a streamlined, user-friendly format, significantly elevating your proficiency in advanced spreadsheet operations.

Further Exploration and Resources

To continue building upon your expertise in advanced data manipulation within Google Sheets, we recommend exploring related tutorials that delve into other complex array operations and data extraction techniques. Understanding the core mechanics of functions like ARRAYFORMULA and TRANSPOSE opens doors to solving numerous other intricate data challenges.