How to Easily Filter Rows by String in Google Sheets Using Query

Authored by

stats writer

December 5, 2025

Understanding the Google Sheets QUERY Function for Data Filtering

The QUERY function in Google Sheets stands as the single most powerful tool for manipulating and analyzing data, offering SQL-like capabilities directly within a spreadsheet environment. Unlike simpler functions like `FILTER` or `VLOOKUP`, **`QUERY`** allows users to perform complex data extraction, aggregation, sorting, and, most importantly for this discussion, sophisticated filtering based on criteria defined in a dedicated query language.

A fundamental requirement in data analysis is the ability to locate records where a cell contains a specific piece of text--a **string** or substring--rather than matching the entire cell value exactly. For instance, you might need to find all customer records based on a partial address or select all product descriptions that include a certain keyword. The solution lies within the **`WHERE`** clause of the QUERY function, utilizing the specialized operator **`CONTAINS`**.

The structure of the query language requires defining the data range, providing the query statement (which must be enclosed in double quotes), and specifying the number of header rows. The power of **`CONTAINS`** is that it operates like a partial match search, scanning every character within the target column's cells to ensure the specified substring is present anywhere within that cell's content. This capability elevates standard spreadsheet filtering to an enterprise-grade data retrieval process.

Core Syntax: Utilizing the CONTAINS Operator

To successfully select rows that include a specific string, you must structure the **`WHERE`** clause to target the correct column and employ the **`CONTAINS`** keyword followed by the desired text, enclosed in single quotes. This is crucial for distinguishing the literal search string from column identifiers or other query commands. The general structure of the query, focusing on string containment, is detailed below:

The standard syntax for selecting rows based on string containment is demonstrated below:

=query(A1:C9, "select A, B where B contains 'this'", 1)

This formulation provides three essential pieces of information to the function. First, the **data range** is `A1:C9`. Second, the **query string** specifies the action: `select A, B where B contains 'this'`. This instructs the function to return columns A and B, but only for those rows where column B contains the specific literal string `'this'`. Finally, the argument **1** indicates that the data range includes one header row, which ensures proper data interpretation by the query engine. Understanding how to correctly quote the column identifiers and the search string is paramount for avoiding parsing errors.

When applying the **`CONTAINS`** operator, remember that the column references (like `A` or `B`) must correspond to the column letters of the input range, not necessarily the column letters of the spreadsheet itself. For example, if your data range starts at `C1`, the first column of your query is referred to as `Col1` or `C`, depending on whether you are using A, B, C notation or Col1, Col2, Col3 notation. For simple column selections starting at A1, using A, B, C is preferred and highly readable. The **`CONTAINS`** operator itself is highly optimized for substring searches, offering an efficient way to filter vast amounts of text data.

Practical Application and Dataset Review

To illustrate the utility of the **`CONTAINS`** operator, we will reference a sample dataset commonly used for sports statistics. This dataset is structured for clarity, allowing us to easily trace how the query filters rows based on partial text matches in the Team column. Effective querying relies on having a stable dataset structure, typically including unique identifiers and category columns.

Our example dataset, shown below, contains three columns: Player Name, Team, and Score. We will focus our filtering criteria exclusively on the Team column to demonstrate inclusion and exclusion based on partial team names. For instance, we will use partial names like 'Lak' to find 'Lakers' or 'Mav' to find 'Mavericks'.

Before executing any query, it is important to confirm the consistency of the data. If the Team column contained typos or inconsistent capitalization, the query results could be inaccurate. However, for the purposes of these examples, we assume the data integrity is sound.

	A	B	C	D	E
1	**Player**	**Team**	**Points**		
2	Andy	Lakers	13.4		
3	Bob	Mavericks	7.8		
4	Carl	Spurs	13.7		
5	Dave	Warriors	22.3		
6	Eric	Mavericks	27.8		
7	Fred	Mavericks	20.8		
8	George	Spurs	12.7		
9	Harold	Lakers	8.2		
10	Isaiah	Warriors	12.5		
11	Joe	Warriors	30.2		
12	Ken	Spurs	22.4		
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					

Example 1: Selecting Based on Contained String (Inclusion)

The most common use case for **`CONTAINS`** is positive filtering--selecting only those rows that explicitly match the search criterion. In our example, we aim to isolate all players associated with the Lakers. Since we are using a partial match, we only need to search for the substring `'Lak'`.

The formula below executes this selection, targeting column B (Team):

We can use the following syntax to select all rows where the Team column contains the string '**Lak**':

F1 =query(A1:C12, "select A, B where B contains 'Lak'", 1)

	A	B	C	D	E	F	G
1	**Player**	**Team**	**Points**			Player	Team
2	Andy	Lakers	13.4			Andy	Lakers
3	Bob	Mavericks	7.8			Harold	Lakers
4	Carl	Spurs	13.7				
5	Dave	Warriors	22.3				
6	Eric	Mavericks	27.8				
7	Fred	Mavericks	20.8				
8	George	Spurs	12.7				
9	Harold	Lakers	8.2				
10	Isaiah	Warriors	12.5				
11	Joe	Warriors	30.2				
12	Ken	Spurs	22.4				
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							

Upon execution, this query successfully identifies and returns the two rows where the Team column contains the string `'Lak'`. This demonstrates the efficiency of **`CONTAINS`** in handling partial matches. The function does not require the cell content to be equal to `'Lak'`, but merely requires `'Lak'` to be a present substring. This approach is superior to using the `=` operator, which would require an exact match to `'Lakers'`, thus failing if the team name was slightly different, such as `'Lakers (West)'`.

Example 2: Excluding Rows Using NOT CONTAINS

When the objective shifts from inclusion to exclusion, the powerful Boolean operator **`NOT`** is utilized alongside **`CONTAINS`**. This combination allows the analyst to select all data records that specifically *do not* meet the specified string criterion, effectively filtering out exceptions or unwanted classifications. This technique is invaluable for isolating remaining data after common categories have been removed.

To achieve this, the **`NOT`** operator must precede the column identifier within the WHERE clause, transforming the condition from one of inclusion to one of exclusion. The syntax below shows how to select all rows where the Team column ***does not*** contain the string `'Lak'`:

F1 =query(A1:C12, "select A, B where not B contains'Lak'", 1)

	A	B	C	D	E	F	G
1	**Player**	**Team**	**Points**			Player	Team
2	Andy	Lakers	13.4			Bob	Mavericks
3	Bob	Mavericks	7.8			Carl	Spurs
4	Carl	Spurs	13.7			Dave	Warriors
5	Dave	Warriors	22.3			Eric	Mavericks
6	Eric	Mavericks	27.8			Fred	Mavericks
7	Fred	Mavericks	20.8			George	Spurs
8	George	Spurs	12.7			Isaiah	Warriors
9	Harold	Lakers	8.2			Joe	Warriors
10	Isaiah	Warriors	12.5			Ken	Spurs
11	Joe	Warriors	30.2				
12	Ken	Spurs	22.4				
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							

Observing the output, the function correctly eliminates all rows containing `'Lak'`, returning every other row in the dataset. This negation is useful for identifying teams other than the Lakers, focusing analysis on the remaining records such as Mavericks, Bulls, and Celtics. The **`NOT CONTAINS`** operation is a staple in data auditing and cleaning processes, ensuring that targeted segments are removed before further processing.

Example 3: Multiple Criteria Using Logical Operators (OR/AND)

In real-world data environments, filtering often requires selecting rows that satisfy one of several possible criteria. The use of Boolean operators **`OR`** and **`AND`** enables the combination of multiple **`CONTAINS`** checks within a single, efficient query statement. Using **`OR`** expands the selection pool, returning a row if *any* of the specified conditions are true.

To select rows where the Team column contains `'Lak'` **OR** the string `'Mav'`, the syntax must link two distinct **`CONTAINS`** conditions. Each condition must be fully specified, repeating the column identifier and the **`CONTAINS`** operator:

F1 =query(A1:C12, "select A, B where B contains 'Lak' or B contains 'Mav'", 1)

	A	B	C	D	E	F	G
1	**Player**	**Team**	**Points**			Player	Team
2	Andy	Lakers	13.4			Andy	Lakers
3	Bob	Mavericks	7.8			Bob	Mavericks
4	Carl	Spurs	13.7			Eric	Mavericks
5	Dave	Warriors	22.3			Fred	Mavericks
6	Eric	Mavericks	27.8			Harold	Lakers
7	Fred	Mavericks	20.8				
8	George	Spurs	12.7				
9	Harold	Lakers	8.2				
10	Isaiah	Warriors	12.5				
11	Joe	Warriors	30.2				
12	Ken	Spurs	22.4				
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							

This compound query accurately returns all records associated with either the Lakers or the Mavericks. Note that if you were to use the **`AND`** operator instead of **`OR`**, the query would likely return zero results, as a single cell cannot simultaneously contain both `'Lak'` and `'Mav'` unless the cell content was something like `'Lak Mav Team'`. Therefore, when filtering for distinct categories, **`OR`** is the required logical connector.

Advanced String Matching: LIKE vs. CONTAINS

Users familiar with standard SQL often look for the **`LIKE`** operator, which is used extensively for pattern matching using wildcards (such as `%` and `_`). It is important to clarify the distinction between **`CONTAINS`** and **`LIKE`** within the Google Sheets query environment, which is based on the Google Visualization API Query Language.

While **`CONTAINS`** performs a straightforward substring search--checking if one string is nested within another--the **`LIKE`** operator offers more granular control using wildcards. For example, if you wanted to find teams whose names start with 'C' and end with 's', you could potentially use `where B like 'C%s'`. In many common cases, **`CONTAINS`** provides the necessary functionality for substring matching, often simplifying the query syntax compared to using **`LIKE`** with explicit wildcards.

However, **`LIKE`** is particularly useful when the position of the substring matters (e.g., finding strings that specifically start or end with a pattern). If you must ensure the selected string appears only at the beginning of a cell, **`LIKE`** or the related **`STARTS WITH`** operator provides better precision than the more permissive **`CONTAINS`** operator. Analysts should choose the operator that best reflects the required matching precision for their specific data task.

Crucial Considerations: Case Sensitivity and Performance Tuning

One of the most critical aspects of string matching in the Google Sheets WHERE clause is case sensitivity. By default, the **`CONTAINS`** operator is **case-sensitive**. This means that a query searching for `'lak'` (lowercase) will fail to match a cell containing `'Lakers'` (capitalized), leading to incomplete results. This behavior is standard but often catches new users off guard, requiring a specific workaround to achieve case-insensitive matching.

To overcome case sensitivity and ensure robust matching regardless of capitalization, you must apply the **`LOWER()`** function to the column identifier within the query string. This transforms all text in the target column to lowercase *before* the comparison takes place. The search string itself must also be written in lowercase. For example, to search for `'lak'` case-insensitively, the syntax would be: `where lower(B) contains 'lak'`. This is a vital technique for handling user-generated input or external data that lacks uniform capitalization.

Finally, when dealing with extremely large datasets (tens of thousands of rows or more), consider the performance implications. Complex queries involving multiple **`OR CONTAINS`** clauses or the use of wrapped functions like **`LOWER()`** can increase calculation time. If speed is a concern, prioritize efficient queries. Instead of chaining many **`OR CONTAINS`** statements, consider normalizing your data first or using a separate helper column with a **`REGEXMATCH`** function, which is sometimes faster for complex pattern matching, before querying the results.