

# How to Extract Unique Rows Using the Google Sheets Query Function

Authored by  
**stats writer**

December 1, 2025

## RECOMMENDED CITATION

stats writer (2025). *How to Extract Unique Rows Using the Google Sheets Query Function*. PSYCHOLOGICAL SCALES. Retrieved from <https://scales.arabpsychology.com/?p=102994>

The process of data cleaning and ensuring data integrity often necessitates the removal of duplicate entries. In Google Sheets, a powerful and efficient way to achieve this highly granular level of uniqueness is by combining the native **Query function** with the specialized **UNIQUE function**. While the `QUERY` function itself is an invaluable tool for filtering and manipulating data using SQL-like clauses, it is the synergistic use of `UNIQUE()` wrapped around the query results that guarantees only distinct, non-redundant rows are returned.

This technique is paramount when dealing with large, complex datasets where multiple columns contribute to the definition of a unique record. Relying solely on the `DISTINCT` keyword within the `QUERY` function can often be insufficient or require complex `GROUP BY` clauses, especially when the goal is to return all columns of the original data, not just aggregated results. By leveraging the `UNIQUE()` wrapper, we establish a robust, straightforward methodology for eliminating duplication across all selected fields simultaneously, thereby streamlining analysis and reporting processes.

Understanding how these two functions interact is essential for any advanced Google Sheets user seeking to optimize their workflow. The combination allows for precise control over the output, ensuring that only necessary and validated information proceeds to the next stage of data processing. This article will provide a comprehensive, step-by-step guide on structuring this crucial formula, complete with practical examples illustrating its effectiveness in various scenarios.

## The Importance of Data Uniqueness in Analysis

In analytical environments, duplicate records can severely skew results, leading to misinformed decisions or inaccurate metrics. For instance, if a sales transaction dataset accidentally records the same purchase twice, any summary report calculating total revenue will be artificially inflated. Eliminating these redundancies is not merely a matter of tidiness; it is a fundamental requirement for maintaining the reliability and validity of any data-driven project. Furthermore, when combining data from multiple sources, it is common to encounter overlapping entries, making a unified methodology for de-duplication indispensable.

While basic manual sorting and filtering can identify obvious duplicates, this approach is impractical for dynamic or large-scale data management. This is where automated solutions, such as the nested `UNIQUE(QUERY(...))` formula, demonstrate their superior efficiency. This combined formula executes complex filtering operations and then applies a definitive uniqueness check across the resulting rows, all within a single cell computation. This method significantly reduces the manual effort involved in data preparation, freeing up analytical resources for interpretation rather than cleaning.

Moreover, structuring data efficiently often requires generating lookup tables or specific reporting views that must contain only unique identifiers or combinations. For example, if you are analyzing product inventory, you might need a list of all unique product category and warehouse

combinations, regardless of how many individual items fall into those categories. The approach detailed below provides the most effective pathway to construct these derived, unique datasets directly within the Google Sheets environment, ensuring that the source data remains untouched while the output is perfectly curated.

## Understanding the Google Sheets Query Function

The **Query function** is often referred to as the most powerful function in Google Sheets because it allows users to manipulate data using a syntax reminiscent of the Structured Query Language (**SQL**). This function enables complex operations such as filtering (using **WHERE** clauses), sorting (using **ORDER BY**), aggregating (using **SUM**, **AVG**, etc.), and grouping (using **GROUP BY**) data contained within a specified range. Its fundamental structure requires three key arguments: the data range, the query string (written in the specialized query language), and the number of headers.

The strength of the **QUERY** function lies in its ability to handle dynamic filtering criteria. For instance, you can select only specific columns, filter rows based on multiple conditional expressions (e.g., date ranges, numerical thresholds, or text matches), and rearrange the output structure completely. However, when the query aims to return all selected columns for every row that matches the criteria, the query language's built-in mechanism for true row-level de-duplication is not as direct as dedicated functions found in traditional database systems. This subtle limitation is precisely why we must integrate the **UNIQUE()** function.

While **QUERY** does support the **SELECT DISTINCT** structure, this keyword only ensures uniqueness for the column specified immediately after it, or it requires the use of **GROUP BY**, which fundamentally changes the output structure into aggregated groups rather than returning the original data rows. If the objective is simply to return the original rows but with exact duplicates removed based on the combination of all selected columns, wrapping the entire query result in **UNIQUE()** is the most accurate and readable solution. This nesting ensures that the powerful selection and filtering capabilities of **QUERY** are utilized first, followed by a definitive uniqueness check on the final result set.

## The Role of the UNIQUE Function in Data Cleaning

The **UNIQUE function** is purpose-built in Google Sheets to identify and return only the distinct rows from a specified range or array. Unlike filtering mechanisms that rely on conditional logic, **UNIQUE()** performs a direct comparison of every row against all others within its input range, ensuring that only the first instance of any identical row combination is retained in the output. This simplicity makes it exceptionally powerful for data hygiene tasks.

When used independently, the `UNIQUE()` function typically takes a simple range reference, such as `=UNIQUE(A1:Z100)`. However, its true potential is unlocked when its input is not a static range but a dynamic array generated by another complex function, such as the `QUERY` result. By receiving the filtered, sorted, or otherwise manipulated data array from the `QUERY` function, `UNIQUE()` acts as the final quality control layer, meticulously scanning the resulting data table for any exact row matches and purging the redundant entries.

Crucially, the `UNIQUE` function evaluates the uniqueness based on the combined values of all cells within a row in the input array. If even a single cell differs between two rows, they are considered unique. This holistic approach to row comparison is what makes it superior for achieving complete row de-duplication after a complex query has been executed. By wrapping the query, we guarantee that the final view presented to the user contains only valid, non-repeating records that satisfy the initial query criteria, thereby fulfilling the requirements for generating a clean and reliable report or analytical dataset.

## Standard Syntax for Extracting Unique Rows

To successfully return only unique rows using the combined functionality, we must nest the `QUERY()` function entirely within the `UNIQUE()` function. This nesting establishes a clear sequence of operations: first, the `QUERY` executes its selection and filtering logic on the source data; second, the output array generated by the `QUERY` becomes the input for the `UNIQUE` function, which then performs the de-duplication. The following basic syntax demonstrates this critical structure:

```
=UNIQUE(QUERY(A1:B16, "SELECT A, B"))
```

In this fundamental structure, the inner `QUERY(A1:B16, "SELECT A, B")` selects columns A and B from the range A1:B16. This operation generates an array containing the selected data, which may still include duplicate rows if identical combinations of values exist across A and B. It is the outer `UNIQUE()` function that then processes this intermediate array, inspecting every row to ensure that the final result set only includes distinct combinations of the values found in columns A and B. This nesting is the definitive technique for achieving unique rows when dynamic selection is also required.

This approach simplifies complex data extraction tasks. By keeping the core filtering logic within the **Query function**, the formula remains highly flexible and readable. If the data range changes or the columns to be selected are altered, only the arguments within the `QUERY` function need modification. The wrapper `UNIQUE()` function guarantees the desired outcome regardless of the complexity of the internal query string, making it a reliable pattern for returning clean, non-redundant views of the source data.

## Practical Example: Filtering Unique Combinations

To illustrate the effectiveness of this nested function structure, let us consider a sample dataset containing information on basketball players. This dataset includes columns for the player's **Team** (Column A) and their **Position** (Column B). Naturally, a team will have multiple players sharing the same position, leading to duplicate combinations of Team and Position in the source data. Our goal is to extract a list of every unique Team-Position combination currently represented in the dataset.

Suppose our initial data structure looks like the image below, containing information for 15 players:

	A	B	C	D
1	<b>Team</b>	<b>Position</b>		
2	A	Guard		
3	A	Guard		
4	A	Guard		
5	A	Forward		
6	A	Forward		
7	B	Guard		
8	B	Forward		
9	B	Forward		
10	B	Forward		
11	B	Center		
12	C	Guard		
13	C	Guard		
14	C	Guard		
15	C	Forward		
16	C	Center		
17				
18				

To perform a query that returns only the unique combinations of **Team** (Column A) and **Position** (Column B) from the range A1:B16, we deploy the combined formula:

```
=UNIQUE(QUERY(A1:B16, "SELECT A, B"))
```

The screenshot below displays the output when this formula is applied in Google Sheets:

D1 fx =UNIQUE(QUERY(A1:B16, "SELECT A, B"))

	A	B	C	D	E
1	<b>Team</b>	<b>Position</b>		Team	Position
2	A	Guard		A	Guard
3	A	Guard		A	Forward
4	A	Guard		B	Guard
5	A	Forward		B	Forward
6	A	Forward		B	Center
7	B	Guard		C	Guard
8	B	Forward		C	Forward
9	B	Forward		C	Center
10	B	Forward			
11	B	Center			
12	C	Guard			
13	C	Guard			
14	C	Guard			
15	C	Forward			
16	C	Center			
17					
18					
19					

Observe the resulting output carefully. The formula successfully processes the input, identifies all redundant rows, and returns only the unique pairings. For instance, the original data might show three different players on Team "A" who all hold the position "Guard." The `UNIQUE(QUERY(...))` combination ensures that the output list contains only one instance of the row "A | Guard," thus providing a definitive list of the existing categories rather than a list of every player entry. This demonstration clearly highlights the functional difference between merely selecting data and ensuring the absolute uniqueness of the resulting records.

## Handling Advanced Queries and Conditional Uniqueness

The versatility of the `UNIQUE(QUERY(...))` structure is further demonstrated when combined with advanced filtering clauses within the **Query function**. Often, analysts need to find unique records only after applying specific criteria to the source data. For example, we might only be interested in the unique Team and Position combinations for players belonging to Team 'A' or Team 'B', intentionally excluding data related to Team 'C'. The nesting structure handles this conditional filtering seamlessly.

To implement this conditional uniqueness, we embed the necessary `WHERE` clause into the `QUERY` string. The `QUERY` function first filters the data based on the condition (e.g., Column A must be 'A'

or 'B'), and only the rows that satisfy this condition are passed to the **UNIQUE function** for deduplication. This two-stage processing ensures both accuracy in filtering and guaranteed uniqueness in the final presentation.

We can use the following advanced query structure to return only unique rows where the team is explicitly equal to 'A' or 'B':

**=UNIQUE(QUERY(A1:B16, "SELECT A, B WHERE A='A' OR A='B'"))**

The subsequent screenshot demonstrates the result of executing this formula, proving that only the data relevant to Teams A and B is considered, and within that subset, only unique pairings of Team and Position are displayed:

D1    fx    =UNIQUE(QUERY(A1:B16, "SELECT A, B WHERE A='A' OR A='B'"))

	A	B	C	D	E
1	<b>Team</b>	<b>Position</b>		A	Guard
2	A	Guard		A	Forward
3	A	Guard		B	Guard
4	A	Guard		B	Forward
5	A	Forward		B	Center
6	A	Forward			
7	B	Guard			
8	B	Forward			
9	B	Forward			
10	B	Forward			
11	B	Center			
12	C	Guard			
13	C	Guard			
14	C	Guard			
15	C	Forward			
16	C	Center			
17					

As confirmed by the output, the resulting array is clean, reflecting only the unique combinations that meet the strict conditional criteria set forth in the inner `QUERY` statement. This capability is vital for generating highly specific reports or creating unique subsets of a larger, unfiltered dataset, ensuring that all subsequent analysis is based on validated, singular entries.

## Integrating Uniqueness with Other Query Clauses

The power of combining `UNIQUE()` with `QUERY()` extends beyond simple selection and filtering; it

can also be used in conjunction with other crucial SQL-like clauses such as `ORDER BY` and `LIMIT`. It is important to remember that all these additional clauses must be contained within the inner `QUERY` string, as the `UNIQUE` function only processes the final array output.

For instance, if you want the unique list of Team and Position pairings, but you require the list to be sorted alphabetically by Team, the `ORDER BY` clause is included in the query string: `"SELECT A, B WHERE A='A' OR A='B' ORDER BY A"`. The `QUERY` function executes the selection, filtering, and sorting sequentially, and only after these operations are complete is the resulting, ordered array fed into `UNIQUE()` for de-duplication. This ensures the output is both unique and presented in the required order.

Similarly, if the dataset is extremely large and you only wish to examine the first few unique results that meet specific criteria, you could incorporate a `LIMIT` clause. For example: `"SELECT A, B WHERE A='A' OR A='B' LIMIT 5"`. In this case, the query finds all matching rows, the `UNIQUE` function removes duplicates from that matching set, and then the `LIMIT` clause restricts the visible output to the first five unique results. This integration demonstrates the flexibility of using the **Query function** as the primary data manipulator, with **UNIQUE function** providing the final, essential layer of data validation.

## Alternative Methods for Ensuring Data Integrity

While the `UNIQUE(QUERY(...))` structure is the most robust and flexible method for dynamic row de-duplication, Google Sheets offers other, more straightforward methods for data integrity maintenance, particularly when dynamic querying is not required. It is useful to understand these alternatives to select the most efficient tool for a given task.

One common alternative is simply applying the `UNIQUE()` function directly to the raw data range. If the entire goal is to obtain a unique list of rows from the original data (e.g., `=UNIQUE(A:Z)`), without any preliminary filtering or column selection, then the simple `UNIQUE()` function is sufficient and more resource-efficient than nesting a `QUERY` function. This method is preferred when the source data is static and the required output is simply a complete, de-duplicated reflection of that source data.

Another related technique involves utilizing the built-in Data Cleaning tools found under the Data menu in Google Sheets. This menu includes features specifically designed to "Remove duplicates." While this feature works well for permanently modifying the source data or generating a copy, it operates as a manual process rather than a dynamic formula. Unlike the formulaic approach which updates automatically when the source data changes, the manual tool requires re-execution every time the data is refreshed. Therefore, for report generation or dynamically updating dashboards, the `UNIQUE(QUERY(...))` formula remains the superior choice due to its

real-time responsiveness and non-destructive nature towards the raw dataset.

## Conclusion: Mastering Dynamic Uniqueness

The ability to dynamically extract and present clean, non-redundant data is a hallmark of sophisticated spreadsheet management. By mastering the combination of the **UNIQUE function** wrapped around the **Query function**, Google Sheets users gain unprecedented control over their output arrays. This technique allows for complex filtering and sorting operations to be executed using the powerful SQL-like syntax, followed by an immediate and guaranteed de-duplication of the resulting rows.

This nested formula structure is indispensable for analytical tasks such as generating master lists of categories, identifying unique customer entries, or preparing data subsets for further processing or visualization. It ensures that regardless of the redundancy present in the source data, the final extracted view is accurate, reliable, and free from errors caused by duplicated records. Adopting this robust methodology will significantly enhance the quality and efficiency of data processing within Google Sheets, moving beyond basic filtering towards advanced, real-time data integrity management.