04 Working with Arrays bq docs -KirkYagami 🧑‍💻 🕵️

https://cloud.google.com/bigquery/docs/arrays🔗

Work with arrays
In GoogleSQL for BigQuery, an array is an ordered list consisting of zero or more values of the same data type. You can construct arrays of simple data types, such as `INT64`, and complex data types, such as `STRUCT`s. The current exception to this is the `ARRAY` data type because arrays of arrays are not supported. To learn more about the `ARRAY` data type, including `NULL` handling, see Array type🔗.

With GoogleSQL, you can construct array literals, build arrays from subqueries using the `ARRAY` function, and aggregate values into an array using the `ARRAY_AGG` function.

You can combine arrays using functions like `ARRAY_CONCAT()`, and convert arrays to strings using `ARRAY_TO_STRING()`.

Accessing array elements

Consider the following emulated table called `Sequences`. This table contains the column `some_numbers` of the `ARRAY` data type.

```
WITH
  Sequences AS (
    SELECT [0, 1, 1, 2, 3, 5] AS some_numbers UNION ALL
    SELECT [2, 4, 8, 16, 32] UNION ALL
    SELECT [5, 10]
  )
SELECT * FROM Sequences;
```

Output:

```
/*--------------------*
 | some_numbers       |
 +--------------------+
 | [0, 1, 1, 2, 3, 5] |
 | [2, 4, 8, 16, 32]  |
 | [5, 10]            |
 *--------------------*/
```

To access array elements in the `some_numbers` column, specify which type of indexing you want to use: either index or `OFFSET(index)` for zero-based indexes, or `ORDINAL(index)` for one-based indexes.

For example:

```
SELECT
  some_numbers,
  some_numbers[0] AS index_0,
  some_numbers[OFFSET(1)] AS offset_1,
  some_numbers[ORDINAL(1)] AS ordinal_1
FROM Sequences;
```

Output:

```
/*--------------------+---------+----------+-----------*
 | some_numbers       | index_0 | offset_1 | ordinal_1 |
 +--------------------+---------+----------+-----------+
 | [0, 1, 1, 2, 3, 5] | 0       | 1        | 0         |
 | [2, 4, 8, 16, 32]  | 2       | 4        | 2         |
 | [5, 10]            | 5       | 10       | 5         |
 *--------------------+---------+----------+-----------*/
```

Note: `OFFSET` and `ORDINAL` will raise errors if the index is out of range. To avoid this, you can use `SAFE_OFFSET` or `SAFE_ORDINAL` to return `NULL` instead of raising an error.

Finding lengths

The `ARRAY_LENGTH` function returns the length of an array.

```
WITH Sequences AS
  (SELECT [0, 1, 1, 2, 3, 5] AS some_numbers
   UNION ALL SELECT [2, 4, 8, 16, 32] AS some_numbers
   UNION ALL SELECT [5, 10] AS some_numbers)
SELECT some_numbers,
       ARRAY_LENGTH(some_numbers) AS len
FROM Sequences;
```

Output:

```
/*--------------------+--------*
 | some_numbers       | len    |
 +--------------------+--------+
 | [0, 1, 1, 2, 3, 5] | 6      |
 | [2, 4, 8, 16, 32]  | 5      |
 | [5, 10]            | 2      |
 *--------------------+--------*/
```

Converting elements in an array to rows in a table

To convert an `ARRAY` into a set of rows, also known as "flattening," use the `UNNEST` operator. `UNNEST` takes an `ARRAY` and returns a table with a single row for each element in the `ARRAY`.

Because `UNNEST` destroys the order of the `ARRAY` elements, you may wish to restore order to the table. To do so, use the optional `WITH OFFSET` clause to return an additional column with the offset for each array element, then use the `ORDER BY` clause to order the rows by their offset.

Example:

```
SELECT *
FROM UNNEST(['foo', 'bar', 'baz', 'qux', 'corge', 'garply', 'waldo', 'fred'])
  AS element
WITH OFFSET AS offset
ORDER BY offset;
```

Output:

```
/*----------+--------*
 | element  | offset |
 +----------+--------+
 | foo      | 0      |
 | bar      | 1      |
 | baz      | 2      |
 | qux      | 3      |
 | corge    | 4      |
 | garply   | 5      |
 | waldo    | 6      |
 | fred     | 7      |
 *----------+--------*/
```

To flatten an entire column of `ARRAY`s while preserving the values of the other columns in each row, use a correlated cross join to join the table containing the `ARRAY` column to the `UNNEST` output of that `ARRAY` column.

With a correlated join, the `UNNEST` operator references the `ARRAY` typed column from each row in the source table, which appears previously in the `FROM` clause. For each row `N` in the source table, `UNNEST` flattens the `ARRAY` from row `N` into a set of rows containing the `ARRAY` elements, and then the cross join joins this new set of rows with the single row `N` from the source table.

Examples:

```
WITH Sequences AS
  (SELECT 1 AS id, [0, 1, 1, 2, 3, 5] AS some_numbers
   UNION ALL SELECT 2 AS id, [2, 4, 8, 16, 32] AS some_numbers
   UNION ALL SELECT 3 AS id, [5, 10] AS some_numbers)
SELECT id, flattened_numbers
FROM Sequences
CROSS JOIN UNNEST(Sequences.some_numbers) AS flattened_numbers;
```

Output:

```
/*------+-------------------*
 | id   | flattened_numbers |
 +------+-------------------+
 |    1 |                 0 |
 |    1 |                 1 |
 |    1 |                 1 |
 |    1 |                 2 |
 |    1 |                 3 |
 |    1 |                 5 |
 |    2 |                 2 |
 |    2 |                 4 |
 |    2 |                 8 |
 |    2 |                16 |
 |    2 |                32 |
 |    3 |                 5 |
 |    3 |                10 |
 *------+-------------------*/
```

Note that for correlated cross joins the `UNNEST` operator is optional and the cross join can be expressed as a comma cross join. Using this shorthand notation, the previous example is consolidated as follows:

```
WITH Sequences AS
  (SELECT 1 AS id, [0, 1, 1, 2, 3, 5] AS some_numbers
   UNION ALL SELECT 2 AS id, [2, 4, 8, 16, 32] AS some_numbers
   UNION ALL SELECT 3 AS id, [5, 10] AS some_numbers)
SELECT id, flattened_numbers
FROM Sequences, Sequences.some_numbers AS flattened_numbers;
```

Output:

```
/*---------+---------------------*
 | id      | flattened_numbers |
 +---------+---------------------+
 |       1 |                   0 |
 |       1 |                   1 |
 |       1 |                   1 |
 |       1 |                   2 |
 |       1 |                   3 |
 |       1 |                   5 |
 |       2 |                   2 |
 |       2 |                   4 |
 |       2 |                   8 |
 |       2 |                  16 |
 |       2 |                  32 |
 |       3 |                   5 |
 |       3 |                  10 |
 *---------+---------------------*/
```

Querying nested arrays

If a table contains an `ARRAY` of `STRUCTs`, you can flatten the `ARRAY` to query the fields of the `STRUCT`. You can also flatten `ARRAY` type fields of `STRUCT` values.

Querying STRUCT elements in an array

The following example uses `UNNEST` with `CROSS JOIN` to flatten an `ARRAY` of `STRUCTs`.

```
WITH Races AS (
  SELECT "800M" AS race,
    [STRUCT("Rudisha" AS name, [23.4, 26.3, 26.4, 26.1] AS laps),
     STRUCT("Makhloufi" AS name, [24.5, 25.4, 26.6, 26.1] AS laps),
     STRUCT("Murphy" AS name, [23.9, 26.0,

 26.8, 26.5] AS laps)] AS participants
)
SELECT
  race,
  participant.name AS name,
  participant.laps[OFFSET(0)] AS lap_1,
  participant.laps[OFFSET(1)] AS lap_2,
  participant.laps[OFFSET(2)] AS lap_3,
  participant.laps[OFFSET(3)] AS lap_4
FROM
  Races, UNNEST(participants) AS participant;
```

Output:

```
/*-------+-----------+-------+-------+-------+-------*
 | race  | name      | lap_1 | lap_2 | lap_3 | lap_4 |
 +-------+-----------+-------+-------+-------+-------+
 | 800M  | Rudisha   | 23.4  | 26.3  | 26.4  | 26.1  |
 | 800M  | Makhloufi | 24.5  | 25.4  | 26.6  | 26.1  |
 | 800M  | Murphy    | 23.9  | 26.0  | 26.8  | 26.5  |
 *-------+-----------+-------+-------+-------+-------*/
```

Querying arrays of arrays

To access elements in an array of arrays, use the `UNNEST` operator multiple times.

```
WITH DoubleArrays AS
  (SELECT [[1, 2, 3], [4, 5, 6]] AS double_array
   UNION ALL SELECT [[0], [NULL]])
SELECT double_array,
       double_array[OFFSET(0)] AS first_array,
       double_array[OFFSET(1)] AS second_array,
       double_array[OFFSET(0)][OFFSET(1)] AS first_array_second_element
FROM DoubleArrays;
```

Output:

```
/*-----------------+--------------+---------------+-------------------------*
 | double_array     | first_array  | second_array  |
first_array_second_element |
 +-----------------+--------------+---------------+-------------------------
+
 | [[1, 2, 3],      | [1, 2, 3]    | [4, 5, 6]     | 2
|
 |  [4, 5, 6]]      |              |               |
|
 | [[0], [NULL]]    | [0]          | [NULL]        | NULL
|
 *-----------------+--------------+---------------+-------------------------
*/
```

To flatten a `STRUCT` that contains an `ARRAY` of `ARRAY`s, use `UNNEST` multiple times.

```
WITH Races AS (
  SELECT "800M" AS race,
    [STRUCT("Rudisha" AS name, [23.4, 26.3, 26.4, 26.1] AS laps),
     STRUCT("Makhloufi" AS name, [24.5, 25.4, 26.6, 26.1] AS laps),
     STRUCT("Murphy" AS name, [23.9, 26.0, 26.8, 26.5] AS laps)] AS
participants
)
SELECT
  race,
  lap
FROM Races,
  UNNEST(participants) AS participant,
  UNNEST(participant.laps) AS lap;
```

Output:

```
/*--------+--------*
 | race   | lap    |
 +--------+--------+
 | 800M   | 23.4   |
 | 800M   | 26.3   |
 | 800M   | 26.4   |
 | 800M   | 26.1   |
 | 800M   | 24.5   |
 | 800M   | 25.4   |
 | 800M   | 26.6   |
 | 800M   | 26.1   |
 | 800M   | 23.9   |
 | 800M   | 26.0   |
 | 800M   | 26.8   |
 | 800M   | 26.5   |
 *--------+--------*/
```

To flatten the same table and retain the `STRUCT` fields, use multiple `UNNEST` operators combined with a correlated join.

```
WITH Races AS (
  SELECT "800M" AS race,
    [STRUCT("Rudisha" AS name, [23.4, 26.3, 26.4, 26.1] AS laps),
     STRUCT("Makhloufi" AS name, [24.5, 25.4, 26.6, 26.1] AS laps),
     STRUCT("Murphy" AS name, [23.9, 26.0, 26.8, 26.5] AS laps)] AS
participants
)
SELECT
  race,
  participant.name AS name,
  lap
FROM Races,
  UNNEST(participants) AS participant,
  UNNEST(participant.laps) AS lap;
```

Output:

```
/*───────+───────────+───────*
 | race  | name      | lap   |
 +───────+───────────+───────+
 | 800M  | Rudisha   | 23.4  |
 | 800M  | Rudisha   | 26.3  |
 | 800M  | Rudisha   | 26.4  |
 | 800M  | Rudisha   | 26.1  |
 | 800M  | Makhloufi | 24.5  |
 | 800M  | Makhloufi | 25.4  |
 | 800M  | Makhloufi | 26.6  |
 | 800M  | Makhloufi | 26.1  |
 | 800M  | Murphy    | 23.9  |
 | 800M  | Murphy    | 26.0  |
 | 800M  | Murphy    | 26.8  |
 | 800M  | Murphy    | 26.5  |
 *───────+───────────+───────*/
```

Filtering array elements

To filter the elements of an array, you can use the `ARRAY` function with a `SELECT` clause.

```
WITH Races AS (
  SELECT "800M" AS race,
    [STRUCT("Rudisha" AS name, [23.4, 26.3, 26.4, 26.1] AS laps),
     STRUCT("Makhloufi" AS name, [24.5, 25.4, 26.6, 26.1] AS laps),
     STRUCT("Murphy" AS name, [23.9, 26.0, 26.8, 26.5] AS laps)] AS
participants
)
SELECT ARRAY(
    SELECT AS STRUCT name, laps
    FROM UNNEST(participants)
    WHERE name LIKE "%Ru%") AS ru_participants
FROM Races;
```

Output:

```
/*------------------------------------------------*
 | ru_participants                                |
 +------------------------------------------------+
 | [{name:"Rudisha", laps:[23.4, 26.3, 26.4, 26.1]}] |
 *------------------------------------------------*/
```

Array Aggregation

You can aggregate values into an array using the `ARRAY_AGG` function. This is particularly useful when you want to group data and return the results as an array.

```
WITH Sales AS (
  SELECT "A" AS store, "2023-01-01" AS date, 200 AS sales
  UNION ALL SELECT "A", "2023-01-02", 300
  UNION ALL SELECT "B", "2023-01-01", 150
  UNION ALL SELECT "B", "2023-01-02", 250
)
SELECT
  store,
  ARRAY_AGG(sales ORDER BY date) AS sales_array
FROM
  Sales
GROUP BY store;
```

Output:

```
/*-------+---------------*
 | store | sales_array   |
 +-------+---------------+
 | A     | [200, 300]    |
 | B     | [150, 250]    |
 *-------+---------------*/
```

In this example, the `ARRAY_AGG` function collects the `sales` values for each store into an array, ordered by date.

Array Concatenation

You can concatenate arrays using the `ARRAY_CONCAT` function.

```
WITH Arrays AS (
  SELECT [1, 2, 3] AS array_1, [4, 5] AS array_2
)
SELECT
```

```
  ARRAY_CONCAT(array_1, array_2) AS concatenated_array
FROM
  Arrays;
```

Output:

```
/*--------------------*
 | concatenated_array |
 +--------------------+
 | [1, 2, 3, 4, 5]    |
 *--------------------*/
```

This concatenates `array_1` and `array_2` into a single array.

Array Length

To find the number of elements in an array, use the `ARRAY_LENGTH` function.

```
WITH Example AS (
  SELECT [1, 2, 3, 4] AS numbers
)
SELECT
  ARRAY_LENGTH(numbers) AS length
FROM
  Example;
```

Output:

```
/*--------*
 | length |
 +--------+
 | 4      |
 *--------*/
```

Array Comparison

You can compare arrays directly in BigQuery to check for equality or to perform other comparisons.

```
WITH Example AS (
  SELECT [1, 2, 3] AS array_1, [1, 2, 3] AS array_2
)
SELECT
  array_1 = array_2 AS arrays_equal
```

```
FROM
  Example;
```

Output:

```
/*──────────────*
 | arrays_equal |
 +──────────────+
 | true         |
 *──────────────*/
```

In this example, `arrays_equal` is `true` because both arrays have the same elements in the same order.

Array Slicing

You can extract a portion of an array using the `ARRAY` function with `OFFSET` and `ORDINAL` ranges.

```
WITH Example AS (
  SELECT [1, 2, 3, 4, 5] AS numbers
)
SELECT
  ARRAY(SELECT number FROM UNNEST(numbers) WHERE number ⩾ 2 AND number ⩽ 4)
AS sliced_array
FROM
  Example;
```

Output:

```
/*────────────────*
 | sliced_array   |
 +────────────────+
 | [2, 3, 4]      |
 *────────────────*/
```

This query returns a slice of the array that includes only the elements between 2 and 4.

Working with Nested Arrays

For nested arrays, you can use multiple `UNNEST` operations to flatten or work with the data.

```
WITH NestedArrays AS (
  SELECT [[1, 2, 3], [4, 5, 6]] AS double_array
)
SELECT
  element
FROM NestedArrays, UNNEST(double_array) AS array, UNNEST(array) AS element;
```

Output:

```
/*---------*
 | element |
 +---------+
 | 1       |
 | 2       |
 | 3       |
 | 4       |
 | 5       |
 | 6       |
 *---------*/
```

This flattens the nested array into a single set of elements.

Combining Arrays with STRUCTs

You can also combine arrays with `STRUCT` types to create more complex data structures.

```
WITH Example AS (
  SELECT "John" AS name, [STRUCT(1 AS id, "Math" AS subject), STRUCT(2 AS id, "Science" AS subject)] AS subjects
)
SELECT
  name,
  subject.id,
  subject.subject
FROM Example, UNNEST(subjects) AS subject;
```

Output:

```
/*-------+----+----------*
 | name  | id | subject   |
 +-------+----+-----------+
 | John  |  1 | Math      |
 | John  |  2 | Science   |
 *-------+----+-----------*/
```

This query unpacks the `STRUCT` array, allowing you to access individual fields within each structure.

Array Union

You can combine the elements of two arrays, removing duplicates, using the `ARRAY` function with `UNION DISTINCT`.

```
WITH Arrays AS (
  SELECT [1, 2, 3] AS array_1, [2, 3, 4] AS array_2
)
SELECT
  ARRAY(SELECT DISTINCT x FROM UNNEST(array_1) AS x UNION DISTINCT SELECT
DISTINCT y FROM UNNEST(array_2) AS y) AS union_array
FROM
  Arrays;
```

Output:

```
/*-----------------*
 | union_array     |
 +-----------------+
 | [1, 2, 3, 4]    |
 *-----------------*/
```

This query combines the elements from both arrays into a single array without duplicates.

Array Intersect

To find common elements between arrays, use `INTERSECT DISTINCT`.

```
WITH Arrays AS (
  SELECT [1, 2, 3] AS array_1, [2, 3, 4] AS array_2
)
SELECT
  ARRAY(SELECT x FROM UNNEST(array_1) AS x INTERSECT DISTINCT SELECT y FROM
UNNEST(array_2) AS y) AS intersect_array
FROM
  Arrays;
```

Output:

```
/*-----------------*
 | intersect_array |
 +-----------------+
  │ [2, 3]          │
  *─────────────────*/
```

This returns the elements that are present in both arrays.