

# Apache Spark Built-in and Higher-Order Functions Examples

## For array types

**array\_distinct(array<T>): array<T>**

Removes duplicate values from the given array.

```
SELECT array_distinct(array(1, 2, 3, null, 3));
```

	array_distinct(array(1, 2, 3, CAST(NULL AS INT), 3)) ▲
1	▶ [1, 2, 3, null]

Showing all 1 rows.



**array\_intersect(array<T>, array<T>): array<T>**

Returns an array of the elements in the intersection of the given two arrays, without duplicates.

```
SELECT array_intersect(array(1, 2, 3), array(1, 3, 5));
```

	array_intersect(array(1, 2, 3), array(1, 3, 5)) ▲
1	▶ [1, 3]

Showing all 1 rows.



## **array\_union(array<T>, array<T>): array<T>**

Returns an array of the elements in the union of the given two arrays, without duplicates.

```
SELECT array_union(array(1, 2, 3), array(1, 3, 5));
```

	array_union(array(1, 2, 3), array(1, 3, 5)) ▲
1	▶ [1, 2, 3, 5]

Showing all 1 rows.



## **array\_except(array<T>, array<T>): array<T>**

Returns an array of the elements in array1 but not in array2, without duplicates.

```
SELECT array_except(array(1, 2, 3), array(1, 3, 5));
```

	array_except(array(1, 2, 3), array(1, 3, 5)) ▲
1	[2]

Showing all 1 rows.



## array\_join(array<String>, String[, String]): String

Concatenates the elements of the given array using the delimiter and an optional string to replace nulls. If no value is set for null replacement, any null value is filtered.

```
SELECT array_join(array('hello', 'world'), ' ');
```

	array_join(array(hello, world), ) ▲
1	hello world

Showing all 1 rows.



```
SELECT array_join(array('hello', null , 'world'), ' ');
```

	<b>array_join(array(hello, CAST(NULL AS STRING), world), )</b> ▲
1	hello world

Showing all 1 rows.



```
SELECT array_join(array('hello', null , 'world'), ' ', ',');
```

	<b>array_join(array(hello, CAST(NULL AS STRING), world), , ,)</b> ▲
1	hello , world

Showing all 1 rows.



## array\_max(array<T>): T

Returns the maximum value in the given array. null elements are skipped.

```
SELECT array_max(array(1, 20, null, 3));
```

	<b>array_max(array(1, 20, CAST(NULL AS INT), 3))</b> ▲
1	20

Showing all 1 rows.



## array\_min(array<T>): T

Returns the minimum value in the given array. null elements are skipped.

```
SELECT array_min(array(1, 20, null, 3));
```

	array_min(array(1, 20, CAST(NULL AS INT), 3)) ▲
1	1

Showing all 1 rows.



## array\_position(array<T>, T): Long

Returns the (1-based) index of the first element of the given array as long.

```
SELECT array_position(array(3, 2, 1), 1);
```

	array_position(array(3, 2, 1), 1) ▲
1	3

Showing all 1 rows.



## **array\_remove(array<T>, T): array<T>**

Remove all elements that equal to the given element from the given array.

```
SELECT array_remove(array(1, 2, 3, null, 3), 3);
```

	array_remove(array(1, 2, 3, CAST(NULL AS INT), 3), 3) ▲
1	▶ [1, 2, null]

Showing all 1 rows.



## **arrays\_overlap(array<T>, array<T>): array<T>**

Returns true if array1 contains at least a non-null element present also in array2. If the arrays have no common element and they are both non-empty and either of them contains a null element null is returned, false otherwise.

```
SELECT arrays_overlap(array(1, 2, 3), array(3, 4, 5));
```

	arrays_overlap(array(1, 2, 3), array(3, 4, 5)) ▲
1	true

Showing all 1 rows.



## array\_sort(array<T>): array<T>

Sorts the input array in ascending order. The elements of the input array must be orderable. Null elements will be placed at the end of the returned array.

```
SELECT array_sort(array('b', 'd', null, 'c', 'a'));
```

	array_sort(array(b, d, CAST(NULL AS STRING), c, a), lambdafunction((IF(((namedlambdavariable() IS NULL) AND (namedlambdavariable() IS NULL)), 0, (IF((namedlambdavariable() IS NULL), 1, (IF((namedlambdavariable() IS NULL), -1, (IF((namedlambdavariable() < namedlambdavariable()), -1, (IF((namedlambdavariable() > namedlambdavariable()), 1, ...
--	--

Showing all 1 rows.



## concat(String, ...): String / concat(array<T>, ...): array<T>

Returns the concatenation of col1, col2, ..., colN.

This function works with strings, binary and compatible array columns.

```
SELECT concat('Spark', 'SQL');
```

	concat(Spark, SQL) ▲
1	SparkSQL

Showing all 1 rows.





```
SELECT concat(array(1, 2, 3), array(4, 5), array(6));
```

	concat(array(1, 2, 3), array(4, 5), array(6)) ▲
1	▶ [1, 2, 3, 4, 5, 6]

Showing all 1 rows.



## **flatten(array<array<T>>): array<T>**

Transforms an array of arrays into a single array.

```
SELECT flatten(array(array(1, 2), array(3, 4)));
```

	flatten(array(array(1, 2), array(3, 4))) ▲
1	▶ [1, 2, 3, 4]

Showing all 1 rows.



## **array\_repeat(T, Int): array<T>**

Returns the array containing element count times.



```
SELECT array_repeat('123', 2);
```

	array_repeat(123, 2) ▲
1	▶["123", "123"]

Showing all 1 rows.



## reverse(String): String / reverse(array<T>): array<T>

Returns a reversed string or an array with reverse order of elements.

```
SELECT reverse('Spark SQL');
```

	reverse(Spark SQL) ▲
1	LQS krapS

Showing all 1 rows.



```
SELECT reverse(array(2, 1, 4, 3));
```

	reverse(array(2, 1, 4, 3)) ▲
1	▶[3, 4, 1, 2]

Showing all 1 rows.



## **sequence(T, T[, T]): array<T>**

Generates an array of elements from start to stop (inclusive), incrementing by step. The type of the returned elements is the same as the type of argument expressions.

```
SELECT sequence(1, 5);
```

	sequence(1, 5) ▲
1	▶ [1, 2, 3, 4, 5]

Showing all 1 rows.



```
SELECT sequence(5, 1);
```

	sequence(5, 1) ▲
1	▶ [5, 4, 3, 2, 1]

Showing all 1 rows.



```
SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval 1 month);
```

	<b>sequence(to_date('2018-01-01'), to_date('2018-03-01'), INTERVAL '1 months')</b> ▲
1	▶ ["2018-01-01", "2018-02-01", "2018-03-01"]

Showing all 1 rows.



## shuffle(array<T>): array<T>

Returns a random permutation of the given array.

```
SELECT shuffle(array(1, 20, 3, 5));
```

	<b>shuffle(array(1, 20, 3, 5))</b> ▲
1	▶ [20, 1, 5, 3]

Showing all 1 rows.



```
SELECT shuffle(array(1, 20, null, 3));
```

	<b>shuffle(array(1, 20, CAST(NULL AS INT), 3))</b> ▲
1	▶ [null, 3, 20, 1]

Showing all 1 rows.





## **slice(array<T>, Int, Int): array<T>**

Subsets the given array starting from index start (or starting from the end if start is negative) with the specified length.

```
SELECT slice(array(1, 2, 3, 4), 2, 2);
```

	slice(array(1, 2, 3, 4), 2, 2) ▲
1	▶ [2, 3]

Showing all 1 rows.



```
SELECT slice(array(1, 2, 3, 4), -2, 2);
```

	slice(array(1, 2, 3, 4), -2, 2) ▲
1	▶ [3, 4]

Showing all 1 rows.



## **array\_zip(array<T>, array<U>, ...): array<struct<T, U, ...>>**

Returns a merged array of structs in which the N-th struct contains all N-th values of input arrays.

```
SELECT arrays_zip(array(1, 2, 3), array(2, 3, 4));
```

	arrays_zip(array(1, 2, 3), array(2, 3, 4)) ▲
1	▶ [{"0": 1, "1": 2}, {"0": 2, "1": 3}, {"0": 3, "1": 4}]

Showing all 1 rows.



```
SELECT arrays_zip(array(1, 2), array(2, 3), array(3, 4));
```

	arrays_zip(array(1, 2), array(2, 3), array(3, 4)) ▲
1	▶ [{"0": 1, "1": 2, "2": 3}, {"0": 2, "1": 3, "2": 4}]

Showing all 1 rows.



## For map types

**map\_from\_arrays(array<K>, array<V>): map<K, V>**

Creates a map with a pair of the given key/value arrays. All elements in keys should not be null.

```
SELECT map_from_arrays(array(1.0, 3.0), array('2', '4'));
```

---

	<code>map_from_arrays(array(1.0, 3.0), array(2, 4))</code> ▲
1	▶{"1.0": "2", "3.0": "4"}

Showing all 1 rows.



## `map_from_entries(array<struct<K, V>>): map<K, V>`

Returns a map created from the given array of entries.

```
SELECT map_from_entries(array(struct(1, 'a'), struct(2, 'b')));
```

	<code>map_from_entries(array(named_struct(col1, 1, col2, a), named_struct(col1, 2, col2, a)))</code> ▲
1	▶{"1": "a", "2": "b"}

Showing all 1 rows.



## `map_concat(map<K, V>, ...): map<K, V>`

Returns the union of all the given maps.

```
SELECT map_concat(map(1, 'a', 2, 'b'), map(3, 'c', 4, 'd'));
```

	<code>map_concat(map(1, a, 2, b), map(3, c, 4, d))</code> ▲
--	---

1	▶{"1": "a", "2": "b", "3": "c", "4": "d"}
---	---

Showing all 1 rows.



## For both array and map types

**element\_at(array<T>, Int): T / element\_at(map<K, V>, K): V**

For arrays, returns an element of the given array at given (1-based) index. If index < 0, accesses elements from the last to the first. Returns null if the index exceeds the length of the array.

For maps, returns a value for the given key, or null if the key is not contained in the map.

```
SELECT element_at(array(1, 2, 3), 2);
```

	element_at(array(1, 2, 3), 2) ▲
1	2

Showing all 1 rows.



```
SELECT element_at(map(1, 'a', 2, 'b'), 2);
```

	▲
--	---

	element_at(map(1, a, 2, b), 2)
1	b

Showing all 1 rows.



## cardinality(array<T>): Int / cardinality(map<K, V>): Int

An alias of size. Returns the size of the given array or a map. Returns -1 if null.

```
SELECT cardinality(array('b', 'd', 'c', 'a'));
```

	cardinality(array(b, d, c, a)) ▲
1	4

Showing all 1 rows.



## Higher-order functions

**transform(array<T>, function<T, U>): array<U> and transform(array<T>, function<T, Int, U>): array<U>**

Transform elements in an array using the function.



If there are two arguments for the lambda function, the second argument means the index of the element.

```
SELECT transform(array(1, 2, 3), x -> x + 1);
```

	transform(array(1, 2, 3), lambdafunction((namedlambdavariable() + 1),
1	▶ [2, 3, 4]

Showing all 1 rows.



```
SELECT transform(array(1, 2, 3), (x, i) -> x + i);
```

	transform(array(1, 2, 3), lambdafunction((namedlambdavariable() + namedlambdavariable()), namedlambdavariable(),
	namedlambdavariable()))

Showing all 1 rows.



## filter(array<T>, function<T, Boolean>): array<T>

Filter the input array using the given predicate.

```
SELECT filter(array(1, 2, 3), x -> x % 2 == 1);
```

	filter(array(1, 2, 3), lambdafunction(((namedlambdavariable() % 2) = 1),
--	--

1	▶ [1, 3]
---	----------

Showing all 1 rows.



## **aggregate(array<T>, A, function<A, T, A>[, function<A, R>]): R**

Apply a binary operator to an initial state and all elements in the array, and reduces this to a single state. The final state is converted into the final result by applying a finish function.

```
SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x);
```

	<b>aggregate(array(1, 2, 3), 0, lambdafunction((namedlambdavariable() + namedlambdavariable()), namedlambdavariable(), namedlambdavariable()), lambdafunction(namedlambdavariable(), namedlambdavariable()))</b>
1	6

Showing all 1 rows.



```
SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x, acc -> acc * 10);
```

	<b>aggregate(array(1, 2, 3), 0, lambdafunction((namedlambdavariable() + namedlambdavariable()), namedlambdavariable(), namedlambdavariable()), lambdafunction((namedlambdavariable() * 10), namedlambdavariable()))</b>
1	60

Showing all 1 rows.



## **exists(array<T>, function<T, Boolean>): Boolean**

Test whether a predicate holds for one or more elements in the array.

```
SELECT exists(array(1, 2, 3), x -> x % 2 == 0);
```

	exists(array(1, 2, 3), lambdafunction(((namedlambdavariable() % 2) = 0),
1	true

Showing all 1 rows.



## **zip\_with(array<T>, array<U>, function<T, U, R>): array<R>**

Merge the two given arrays, element-wise, into a single array using function. If one array is shorter, nulls are appended at the end to match the length of the longer array, before applying function.

```
SELECT zip_with(array(1, 2, 3), array('a', 'b', 'c'), (x, y) -> (y, x));
```

	zip_with(array(1, 2, 3), array(a, b, c), lambdafunction(named_struct(y, namedlambdavariable(), x,
	namedlambdavariable()), namedlambdavariable(), namedlambdavariable()))

Showing all 1 rows.



```
SELECT zip_with(array(1, 2), array(3, 4), (x, y) -> x + y);
```

	<code>zip_with(array(1, 2), array(3, 4), lambdafunction((namedlambdavariable() + namedlambdavariable()), namedlambdavariable(), namedlambdavariable()))</code>	▲
--	--	---

Showing all 1 rows.



	<code>zip_with(array(a, b, c), array(d, e, f), lambdafunction(concat(namedlambdavariable(), namedlambdavariable()), namedlambdavariable(), namedlambdavariable()))</code>	▲
--	---	---

Showing all 1 rows.

