

# fn make\_expr\_count

Michael Shoemate

June 30, 2025

This document proves that the implementation of `make_expr_count` in `mod.rs` at commit `f5bb719` (outdated<sup>1</sup>) satisfies its proof definition.

## 1 Hoare Triple

### Preconditions

#### Compiler-verified

- Argument `input_domain` of type `ExprDomain`
- Argument `input_metric` of type `PartitionDistance<MI>`
- Argument `expr` of type `Expr`
- Generic `MI` must implement `UnboundedMetric`
- Const generic `P` must be of type `usize`
- `(ExprDomain, PartitionDistance<MI>)` implements `MetricSpace`
- `(ExprDomain, LpDistance<P, f64>)` implements `MetricSpace`
- `Expr` implements `StableExpr<PartitionDistance<MI>, PartitionDistance<MI>`

#### Caller-verified

None

### Pseudocode

```
1 def make_expr_count(  
2     input_domain: ExprDomain, input_metric: PartitionDistance[MI], expr: Expr  
3 ) -> Transformation:  
4     match expr: #  
5         case Agg(Count(input, include_nulls)):  
6             if include_nulls:  
7                 strategy = Strategy.Len  
8             else:  
9                 strategy = Strategy.Count  
10  
11         case Function(inputs, function=FunctionExpr.NullCount):  
12             (input,) = inputs  
13             strategy = Strategy.NullCount
```

---

<sup>1</sup>See new changes with `git diff f5bb719..3fc1a47 rust/src/transformations/make_stable_expr/expr_count/mod.rs`

```

14
15     case Agg(NUnique(input)):
16         strategy = Strategy.NUnique
17
18     case _:
19         raise ValueError("expected count, null_count, len, or n_unique expression")
20
21 # check if input is row-by-row
22 is_row_by_row = input.make_stable(input_domain.as_row_by_row(), input_metric).is_ok() #
23
24 # construct prior transformation
25 t_prior = input.make_stable(input_domain, input_metric) #
26 middle_domain, middle_metric = t_prior.output_space()
27
28 by, margin = middle_domain.context.grouping("count") #
29
30 output_domain = ExprDomain.new( #
31     column=SeriesDomain.new( #
32         middle_domain.column.name,
33         AtomDomain.default(u32),
34     ),
35     context=Context.Grouping(
36         by=by,
37         margin=Margin(
38             max_partition_length=1,
39             max_num_partitions=margin.max_num_partitions,
40             max_partition_contributions=None,
41             max_influenced_partitions=margin.max_influenced_partitions,
42             public_info=margin.public_info,
43         ),
44     )
45 )
46
47 match strategy: #
48     case Strategy.Len:
49         will_count_all = is_row_by_row
50     case Strategy.Count:
51         will_count_all = is_row_by_row and not middle_domain.column.nullable
52     case _:
53         will_count_all = False
54
55 public_info = margin.public_info if will_count_all else None #
56
57 def function(e: Expr) -> Expr: #
58     match strategy:
59         case Strategy.Count:
60             return e.count()
61         case Strategy.NullCount:
62             return e.null_count()
63         case Strategy.Len:
64             return e.len()
65         case Strategy.NUnique:
66             return e.n_unique()
67
68 return t_prior >> Transformation.new( #
69     middle_domain,
70     output_domain, #
71     Function.then_expr(function),
72     middle_metric,
73     LpDistance.default(),
74     counting_query_stability_map(public_info), #
75 )

```

## Postcondition

**Theorem 1.1.** For every setting of the input parameters (`input_domain`, `input_metric`, `expr`, `MI`, `P`) to `make_expr_count` such that the given preconditions hold, `make_expr_count` raises an exception (at compile time or run time) or returns a valid transformation. A valid transformation has the following properties:

1. (Appropriate output domain). For every element  $x$  in `input_domain`, `function(x)` is in `output_domain` or raises a data-independent runtime exception.
2. (Stability guarantee). For every pair of elements  $x, x'$  in `input_domain` and for every pair  $(d_{in}, d_{out})$ , where  $d_{in}$  has the associated type for `input_metric` and  $d_{out}$  has the associated type for `output_metric`, if  $x, x'$  are  $d_{in}$ -close under `input_metric`, `stability_map(d_in)` does not raise an exception, and `stability_map(d_in) ≤ d_out`, then `function(x), function(x')` are  $d_{out}$ -close under `output_metric`.

## 2 Proof

Starting from line 4, `expr` is analyzed to determine the type of counting query `strategy` and input to the counting query `input`.

All preconditions for `make_stable` on line 25 are compiler-verified, therefore by the postcondition `t_prior` is a valid transformation.

To prove that the output is a valid transformation, we must first prove that the transformation on line 68 is valid.

*Proof. Appropriate Output Domain* Since the count transformation is not row-by-row, line 28 disallows this count transformation from being constructed in a row-by-row context, satisfying the requirements of `ExprDomain`.

By the definition of each of the allowed counting expressions in 4, the resulting output will always contain one series whose name matches the active series' name, and the data type of elements in the series is `u32`, which is consistent with 31. This series is then part of a dataframe and operations are continued to be applied with the same context, on line 30. This domain is then used on line 70 to construct the count transformation.  $\square$

### *Proof. Stability Guarantee*

To determine if `input` is row-by-row, line 22 checks if `input` can be parsed into a row-by-row transformation.

Line 47 uses domain descriptors to determine if the counting query will count all rows. This is the case if the counting query is `Len`, or if the counting query counts all non-null values in data that doesn't contain nulls.

If all rows are to be counted, then the `public_info` domain descriptor about the partitioning applies to this count query strategy, on line 55.

The function defined on line 57 applies the counting query to the input expression with the given query strategy. The stability of this function is governed by the stability map returned by `counting_query_stability_map`, as the `Len`, `Count` and `NullCount` strategies can be expressed as arbitrary predicates, and the `NUnique` strategy can be considered as a 1-stable (non-row-by-row) "unique" transformation chained with a counting query with the `Len` strategy.  $\square$

Since it has been shown that both `t_prior` and the count transformation are valid transformations, then the preconditions for `make_chain_tt` are met (invoked via the right-shift operator shorthand).