`fn make_expr_count`

Michael Shoemate

August 27, 2025

This document proves that the implementation of `make_expr_count` in `mod.rs` at commit f5bb719 (outdated[1]) satisfies its proof definition.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

```python
def make_expr_count(
    input_domain: ExprDomain, input_metric: PartitionDistance[MI], expr: Expr
) -> Transformation:
    match expr:  #
        case Agg(Count(input, include_nulls)):
            if include_nulls:
                strategy = Strategy.Len
            else:
                strategy = Strategy.Count

        case Function(inputs, function=FunctionExpr.NullCount):
            (input,) = inputs
            strategy = Strategy.NullCount
```

[1] See new changes with `git diff f5bb719..b7ff303 rust/src/transformations/make_stable_expr/expr_count/mod.rs`

```python

        case Agg(NUnique(input)):
            strategy = Strategy.NUnique

        case _:
            raise ValueError("expected count, null_count, len, or n_unique expression")

    # check if input is row-by-row
    is_row_by_row = input.make_stable(input_domain.as_row_by_row(), input_metric).is_ok()  #


    # construct prior transformation
    t_prior = input.make_stable(input_domain, input_metric)  #
    middle_domain, middle_metric = t_prior.output_space()

    by, margin = middle_domain.context.grouping("count")  #

    output_domain = ExprDomain.new(  #
        column=SeriesDomain.new(  #
            middle_domain.column.name,
            AtomDomain.default(u32),
        ),
        context=Context.Grouping(
            by=by,
            margin=Margin(
                max_partition_length=1,
                max_num_partitions=margin.max_num_partitions,
                max_partition_contributions=None,
                max_influenced_partitions=margin.max_influenced_partitions,
                public_info=margin.public_info,
            ),
        )
    )

    match strategy:  #
        case Strategy.Len:
            will_count_all = is_row_by_row
        case Strategy.Count:
            will_count_all = is_row_by_row and not middle_domain.column.nullable
        case _:
            will_count_all = False

    public_info = margin.public_info if will_count_all else None  #

    def function(e: Expr) -> Expr:  #
        match strategy:
            case Strategy.Count:
                return e.count()
            case Strategy.NullCount:
                return e.null_count()
            case Strategy.Len:
                return e.len()
            case Strategy.NUnique:
                return e.n_unique()

    return t_prior >> Transformation.new(  #
        middle_domain,
        output_domain,  #
        Function.then_expr(function),
        middle_metric,
        LpDistance.default(),
        counting_query_stability_map(public_info),  #
    )
```

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 error (at compile time or run time) or returns a valid transformation. A valid transformation has the following properties:

1. (Data-independent runtime errors). For every pair of members x and x' in `input_domain`, `invoke`(x) and `invoke`(x') either both return the same error or neither return an error.

2. (Appropriate output domain). For every member x in `input_domain`, `function`(x) is in `output_domain` or raises a data-independent runtime error.

3. (Stability guarantee). For every pair of members x and 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 error, 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 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. □

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. □

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).