

# fn make\_private\_group\_by

Michael Shoemate

This proof resides in “**contrib**” because it has not completed the vetting process.

Proves soundness of `fn make_private_group_by` in `mod.rs` at commit `0db9c6036` (outdated<sup>1</sup>).

## 1 Hoare Triple

### Precondition

#### Compiler-verified

- Generic MI must implement trait `UnboundedMetric`.
- Generic MO must implement trait `ApproximateMeasure`.

#### User-verified

None

### Pseudocode

```
1 def make_private_group_by(
2     input_domain: DslPlanDomain,
3     input_metric: FrameDistance[MI],
4     output_measure: MO,
5     plan: DslPlan,
6     global_scale: Option[f64],
7     threshold: Option[u32],
8 ):
9     input, group_by, aggs, key_sanitizer = match_group_by(plan) #
10
11    # 1: establish stability of 'group_by'
12    t_prior = input.make_stable(input_domain, input_metric) #
13    middle_domain, middle_metric = t_prior.output_space()
14
15    for expr in group_by: #
16        # grouping keys must be stable
17        t_group_by = expr.make_stable(
18            WildExprDomain(
19                columns=middle_domain.series_domains,
20                context=ExprContext.RowByRow,
21            ),
22            LOPInfDistance(middle_metric[0]),
23        )
24
25        series_domain = t_group_by.output_domain.column
```

<sup>1</sup>See new changes with `git diff 0db9c6036..46e01e0 rust/src/measurements/make_private_lazyframe/group_by/mod.rs`

```

26     try:
27         domain = series_domain.element_domain(CategoricalDomain)
28     except Exception:
29         pass
30
31     if domain is not None and domain.categories() is None:
32         raise "Categories are data-dependent, which may reveal sensitive record ordering
33 ."
34
35 group_by_id = HashSet.from_iter(group_by) #
36 margin = middle_domain.get_margin(group_by_id) #
37
38 # 2: prepare for release of 'aggs'
39 match key_sanitizer:
40     case KeySanitizer.Join(keys):
41         num_keys = LazyFrame.from_(keys).select([len()]).collect()
42         margin.max_num_partitions = num_keys.column("len").u32().last() #
43         is_join = True
44     case _:
45         is_join = False
46
47 m_expr_aggs = [
48     make_private_expr(
49         WildExprDomain( #
50             columns=middle_domain.series_domains,
51             context=ExprContext.Aggregation(margin),
52         ),
53         PartitionDistance(middle_metric),
54         output_measure,
55         expr,
56         global_scale,
57     ) for expr in aggs
58 ]
59 m_aggs = make_composition(m_expr_aggs)
60
61 f_comp = m_aggs.function
62 f_privacy_map = m_aggs.privacy_map
63
64 # 3: prepare for release of 'keys'
65 dp_exprs, null_exprs = zip(*((plan.expr, plan.fill) for plan in m_aggs.invoke(input)))
66
67 # 3.2: reconcile information about the threshold
68 if margin.invariant is not None or is_join: #
69     threshold_info = None
70 elif match_filter(key_sanitizer) is not None: #
71     name, threshold_value = match_filter(key_sanitizer)
72     noise = find_len_expr(dp_exprs, name)[1]
73     threshold_info = name, noise, threshold_value, False
74 elif threshold is not None: #
75     name, noise = find_len_expr(dp_exprs, None)
76     threshold_info = name, noise, threshold, True
77 else: #
78     raise f"The key set of {group_by_id} is private and cannot be released."
79
80 # 3.3: update key sanitizer
81 if threshold_info is not None: #
82     name, _, threshold_value, is_present = threshold_info
83     threshold_expr = col(name).gt(lit(threshold_value))
84     if not is_present and predicate is not None: #
85         key_sanitizer = KeySanitizer.Filter(threshold_expr.and_(predicate))
86     else:
87         key_sanitizer = KeySanitizer.Filter(threshold_expr)
88
89 elif isinstance(key_sanitizer, KeySanitizer.Join): #

```

```

89     key_sanitizer.fill_null = []
90     for dp_expr, null_expr in zip(dp_exprs, null_exprs):
91         name = dp_expr.meta().output_name()
92         if null_expr is None:
93             raise f"fill expression for {name} is unknown"
94
95     key_sanitizer.fill_null.append(col(name).fill_null(null_expr))
96
97 # 4: build final measurement
98 def function(arg: DslPlan) -> DslPlan: #
99     output = DslPlan.GroupBy(
100         input=arg,
101         keys=group_by,
102         aggs=[p.expr for p in f_comp.eval(arg)],
103         apply=None,
104         maintain_order=False,
105     )
106     match key_sanitizer:
107         case KeySanitizer.Filter(predicate):
108             output = DslPlan.Filter(input=output, predicate=predicate)
109         case KeySanitizer.Join(
110             how,
111             left_on,
112             right_on,
113             options,
114             fill_null,
115             keys=labels,
116         ):
117             match how: #
118                 case JoinType.Left:
119                     input_left, input_right = labels, output
120                 case JoinType.Right:
121                     input_left, input_right = output, labels
122                 case _:
123                     raise "unreachable"
124
125             output = DslPlan.HStack(
126                 input=DslPlan.Join(
127                     input_left,
128                     input_right,
129                     left_on,
130                     right_on,
131                     options,
132                     predicates=[],
133                 ),
134                 exprs=fill_null,
135                 options=ProjectionOptions.default(),
136             )
137     return output
138
139 def privacy_map(d_in: Bounds): #
140     bound = d_in.get_bound(group_by_id)
141
142     # incorporate all information into optional bounds
143     l0 = option_min(bound.num_groups, margin.max_groups)
144     li = option_min(bound.per_group, margin.max_length)
145     l1 = d_in.get_bound(HashSet.new()).per_group  #
146
147     # reduce optional bounds to concrete bounds
148     if l0 is not None and l1 is not None and li is not None:
149         pass
150     elif l1 is not None:
151         l0 = l0 or l1 #
152         li = li or l1 #

```

```

153     elif 10 is not None and li is not None:
154         l1 = 10.inf_mul(li) #
155     else: #
156         raise f"num_groups ({10}), total contributions ({l1}), and per_group ({li}) are
157         not sufficiently well-defined."
158
159     d_out = f_privacy_map.eval((10, l1, li))
160
161     if margin.invariant is not None or is_join: #
162         pass
163     elif threshold_info is not None: #
164         _, noise, threshold_value, _ = threshold_info
165         if li >= threshold_value:
166             raise f"Threshold must be greater than {li}."
167
168         d_instability = threshold_value.neg_inf_sub(li)
169         delta_single = integrate_discrete_noise_tail(
170             noise.distribution, noise.scale, d_instability
171         )
172         delta_joint = (1).inf_sub(
173             (1).neg_inf_sub(delta_single).neg_inf_powi(IBig.from_(10))
174         )
175         d_out = M0.add_delta(d_out, delta_joint)
176     else:
177         raise "the key-set is sensitive"
178
179     return d_out
180
181     return t_prior >> Measurement.new(
182         middle_domain,
183         function,
184         middle_metric,
185         output_measure,
186         privacy_map,
187     )

```

## Postconditions

**Theorem 1.1.** For every setting of the input parameters (`input_domain`, `input_metric`, `output_measure`, `plan`, `global_scale`, `threshold`, `MI`, `M0`) to `make_private_group_by` such that the given preconditions hold, `make_private_group_by` raises an error (at compile time or run time) or returns a valid measurement. A valid measurement 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. (Privacy 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_measure`, if  $x, x'$  are `d_in`-close under `input_metric`, `privacy_map(d_in)` does not raise an error, and `privacy_map(d_in) = d_out`, then `function(x), function(x')` are `d_out`-close under `output_measure`.

## 2 Proof

We now prove the postcondition (Theorem 1.1).

*Proof.* The function logic breaks down into parts:

1. establish stability of group by (line 11)

2. prepare for release of `aggs` (line 37)
3. prepare for release of `keys` (line 63)
  - (a) reconcile information about the threshold (line 66)
  - (b) update key sanitizer (line 79)
4. build final measurement (line 97)
  - (a) construct function (line 98)
  - (b) construct privacy map (line 139)

`match_group_by` on line 9 returns `input` (the input plan), `group_by` (the grouping keys), `aggs` (the list of expressions to compute per-partition), and `key_sanitizer` (details on how to sanitize the key-set).

## 2.1 Stability of grouping

By the postcondition of `StableDslPlan.make_stable`, `t_prior` is a valid transformation (line 12).

The loop on line 15 ensures that each column in `group_by` is stable, and that the encoding of data in each group-by column is not data-dependent. Therefore data is grouped in a stable manner, with no data-dependent encoding or exceptions.

## 2.2 Prepare to release aggs

`margin` denotes what is considered public information about the key set, pulled from descriptors in the input domain (line 34). An upper bound on the total number of groups can be statically derived via the length of the public keys in the join. Line 41 retrieves this information from the public keys and assigns it to the margin. `is_join` indicates that key sanitization will occur via a join.

Line 48 starts constructing a joint measurement for releasing the per-partition aggregations `aggs`. Each measurement's input domain is the wildcard expression domain, used to prepare computations that will be applied over data grouped by `group_by`.

By the postcondition of `make_basic_composition`, `m_exprs` is a valid measurement that prepares a batch of expressions that, when executed via `f_comp`, satisfies the privacy guarantee of `f_privacy_map`.

Now that we've prepared the necessary prerequisites for privatizing the aggregations, we switch to privatizing the keys.

## 2.3 Prepare to release keys

`key_sanitization` needs to be updated with information that was not available in the initial match on line 9.

- When joining, we need expressions for filling null values corresponding to partitions that don't exist in the sensitive data.
- When filtering, a threshold may be passed into the constructor, and we must determine a suitable column to filter/threshold against.

By the definition of `m_aggs`, invocation returns a list of expressions and fill expressions. These will be used for the filtering sanitization and join sanitization, respectively.

### 2.3.1 Reconcile information when filtering

Line 66 reconciles the threshold information.

- In the setting where grouping keys are considered public, or key sanitization is handled via a join, no thresholding is necessary (line 67).
- Otherwise, if the key sanitizer contains filtering criteria (line 69), then by the postcondition of `find_len_expr`, filtering on `name` can be used to satisfy  $\delta$ -approximate DP. `noise` of type `NoisePlugin` details the noise distribution and scale. `threshold_info` then contains the column name, noise distribution, threshold value and whether a filter needs to be inserted into the query plan. In this case, since the threshold comes from the query plan, it is not necessary to add it to the query plan, and is therefore false.
- In the case that a threshold has been provided to the constructor (line 73), then `find_len_expr` will search for a suitable column to threshold on, returning with the `name` and `noise` distribution of the column. Since the threshold comes from the constructor and not the plan, it will be necessary to add this filtering threshold to the query plan (explaining the true value).
- By line 76 no suitable filtering criteria have been found, and by the first case there is no suitable invariant for the margin or explicit join keys, so it is not possible to release the keys in a way that satisfies differential privacy, and the constructor refuses to build a measurement.

In common use through the context API, if a mechanism is allotted a delta parameter for stable key release but doesn't already satisfy approximate-DP, then a search is conducted for the smallest suitable threshold parameter. The branching logic from line 66 is intentionally written to ignore the constructor threshold when a suitable filtering threshold is already detected in the plan, to avoid overwriting/changing it.

### 2.3.2 Update key sanitizer

We now update `key_sanitization` starting from line 79:

- When filtering (line 80), `threshold_info` will always be set. `threshold_expr` reflects the reconciled criteria, using the chosen filtering column and threshold. This threshold expression is applied either way the logic branches on line 83. The first case preserves any additional filtering criteria that was already present in the plan, but not used for key release.
- When joining (line 88) the sanitizer needs a way to fill missing values from partitions missing in the data. This is provided by `null_exprs`, which contain imputation strategies for filling in missing values in a way that is indistinguishable from running the mechanism on an empty partition.

`key_sanitizer` now contains all necessary information to ensure that the keys are sanitized, and will be used to construct the function. `threshold_info` and `is_join` are consistent with `key_sanitizer`, and will be used to construct the privacy map.

## 2.4 Build final measurement

### 2.4.1 Function

Line 98 builds the function of the measurement, using all of the properties proven of the variables established thus far. The function returns a `DslPlan` that applies each expression from `m_exprs` to `arg` grouped by `keys`. `key_sanitizer` is conveyed into the plan, if set, to ensure that the keys are also privatized if necessary.

In the case of the join privatization, by the definition of `KeySanitizer`, the join will either be a left or right join. The branching swaps the input plan and labels plan to ensure that the sensitive input data is always joined against the labels, but using the same join type as in the original plan. Once the join is applied, the fill imputation expressions are applied, hiding which partitions don't exist in the original data.

It is assumed that the emitted DSL is executed in the same fashion as is done by Polars. This proof/implementation does not take into consideration side-channels involved in the execution of the DSL.

### 2.4.2 Privacy Map

Line 139 builds the privacy map of the measurement. The measurement for each expression expects data set distances in terms of a triple:

- $L^0$ : the greatest number of groups that can be influenced by any one individual. This is bounded above by `bound.num_groups` and more loosely by `margin.max_groups`, but can also be bounded by the  $L^1$  distance on line 151.
- $L^\infty$ : the greatest number of records that can be added or removed by any one individual in each partition. This is bounded above by `bound.per_group` and more loosely by `margin.max_length`, but can also be bounded by the  $L^1$  distance on line 152.
- $L^1$ : the greatest total number of records that can be added or removed across all partitions. This is bounded by per-group contributions when all data is in one group on line 145, but can also be bounded by the product of the  $L^0$  and  $L^\infty$  bounds on line 154.

By the postcondition of `f_privacy_map`, the privacy loss of releasing the output of `aggs`, when grouped data sets may differ by this distance triple, is `d_out`.

We also need to consider the privacy loss from releasing `keys`. On line 160 under the `public_info` invariant, or under the join sanitization, releases on any neighboring datasets  $x$  and  $x'$  will share the same key-set, resulting in zero privacy loss.

We now adapt the proof from [Rog23] (Theorem 7) to consider the case of stable key release from line 162. Consider  $S$  to be the set of labels that are common between  $x$  and  $x'$ . Define event  $E$  to be any potential outcome of the mechanism for which all labels are in  $S$  (where only stable partitions are released). We then lower bound the probability of the mechanism returning an event  $E$ . In the following,  $c_j$  denotes the exact count for partition  $j$ , and  $Z_j$  is a random variable distributed according to the distribution used to release a noisy count.

$$\begin{aligned} \Pr[E] &= \prod_{j \in x \setminus x'} \Pr[c_j + Z_j \leq T] \\ &\geq \prod_{j \in x \setminus x'} \Pr[\Delta_\infty + Z_j \leq T] \\ &\geq \Pr[\Delta_\infty + Z_j \leq T]^{\Delta_0} \end{aligned}$$

The probability of returning a set of stable partitions ( $\Pr[E]$ ) is the probability of not returning any of the unstable partitions. We now solve for the choice of threshold  $T$  such that  $\Pr[E] \geq 1 - \delta$ .

$$\begin{aligned} \Pr[\Delta_\infty + Z_j \leq T]^{\Delta_0} &= \Pr[Z_j \leq T - \Delta_\infty]^{\Delta_0} \\ &= (1 - \Pr[Z_j > T - \Delta_\infty])^{\Delta_0} \end{aligned}$$

Let `d_instability` denote the distance to instability of  $T - \Delta_\infty$ . By the postcondition of `integrate_discrete_noise_tail`, the probability that a random noise sample exceeds `d_instability` is at most `delta_single`. Therefore  $\delta = 1 - (1 - \text{delta\_single})^{\Delta_0}$ . This gives a probabilistic-DP or probabilistic-zCDP guarantee, which implies approximate-DP or approximate-zCDP guarantees respectively. This privacy loss is then added to `d_out`.

Together with the potential increase in delta for the release of the key set, then it is shown that `function(x)`, `function(x')` are `d_out`-close under `output_measure`. □

## References

- [Rog23] Ryan Rogers. A unifying privacy analysis framework for unknown domain algorithms in differential privacy, 2023.