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Proves soundness of `fn make_private_group_by` in `mod.rs` at commit 0db9c6036 (outdated[1]).

1 Hoare Triple

Precondition

Compiler-verified

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

User-verified

None

Pseudocode

```python
def make_private_group_by(
    input_domain: DslPlanDomain,
    input_metric: FrameDistance[MI],
    output_measure: MO,
    plan: DslPlan,
    global_scale: Optional[f64],
    threshold: Optional[u32],
):
    input, group_by, aggs, key_sanitizer = match_group_by(plan) #

    # 1: establish stability of `group_by`
    t_prior = input.make_stable(input_domain, input_metric)  #
    middle_domain, middle_metric = t_prior.output_space()

    for expr in group_by: #
        # grouping keys must be stable
        t_group_by = expr.make_stable(
            WildExprDomain(
                columns=middle_domain.series_domains,
                context=ExprContext.RowByRow,
            ),
            L0PInfDistance(middle_metric[0]),
```

[1] See new changes with `git diff 0db9c6036..53707708 rust/src/measurements/make_private_lazyframe/group_by/mod.rs`

```python
        )

        series_domain = t_group_by.output_domain.column
        try:
            domain = series_domain.element_domain(CategoricalDomain)
        except:
            pass

        if domain is not None and domain.categories() is None:
            raise f"Categories are data-dependent, which may reveal sensitive record ordering."

    group_by_id = HashSet.from_iter(group_by)  #
    margin = middle_domain.get_margin(group_by_id)  #

    # 2: prepare for release of ‘aggs‘
    m_expr_aggs = [
        make_private_expr(
            WildExprDomain(  #
                columns=middle_domain.series_domains,
                context=ExprContext.Aggregation(margin),
            ),
            PartitionDistance(middle_metric),
            output_measure,
            expr,
            global_scale,
        ) for expr in aggs
    ]
    m_aggs = make_composition(m_expr_aggs)

    f_comp = m_aggs.function
    f_privacy_map = m_aggs.privacy_map

    # 3: prepare for release of ‘keys‘
    dp_exprs, null_exprs = zip(*((plan.expr, plan.fill) for plan in m_aggs.invoke(input)))

    # 3.1: reconcile information about the join
    match key_sanitizer:
        case KeySanitizer.Join(keys):
            num_keys = LazyFrame.from_(keys).select([len()]).collect()
            margin.max_num_partitions = num_keys.column("len").u32().last()  #
            is_join = True
        case _:
            is_join = False

    # 3.2: reconcile information about the threshold
    if margin.invariant is not None or is_join:  #
        threshold_info = None
    elif match_filter(key_sanitizer) is not None:  #
        name, threshold_value = match_filter(key_sanitizer)
        noise = find_len_expr(dp_exprs, name)[1]
        threshold_info = name, noise, threshold_value, False
    elif threshold is not None:  #
        name, noise = find_len_expr(dp_exprs, None)
        threshold_info = name, noise, threshold, True
    else:  #
        raise f"The key set of {group_by_id} is private and cannot be released."

    # 3.3: update key sanitizer
    if threshold_info is not None:  #
        name, _, threshold_value, is_present = threshold_info
        threshold_expr = col(name).gt(lit(threshold_value))
        if not is_present and predicate is not None:  #
            key_sanitizer = KeySanitizer.Filter(threshold_expr.and_(predicate))
```

```
        else:
            key_sanitizer = KeySanitizer.Filter(threshold_expr)

    elif isinstance(key_sanitizer, KeySanitizer.Join):  #
        key_sanitizer.fill_null = []
        for dp_expr, null_expr in zip(dp_exprs, null_exprs):
            name = dp_expr.meta().output_name()
            if null_expr is None:
                raise f"fill expression for {name} is unknown"

            key_sanitizer.fill_null.append(col(name).fill_null(null_expr))

    # 4: build final measurement
    def function(arg: DslPlan) -> DslPlan:  #
        output = DslPlan.GroupBy(
            input=arg,
            keys=group_by,
            aggs=[p.expr for p in f_comp.eval(arg)],
            apply=None,
            maintain_order=False,
        )
        match key_sanitizer:
            case KeySanitizer.Filter(predicate):
                output = DslPlan.Filter(input=output, predicate=predicate)
            case KeySanitizer.Join(
                how,
                left_on,
                right_on,
                options,
                fill_null,
                keys=labels,
            ):
                match how:  #
                    case JoinType.Left:
                        input_left, input_right = labels, output
                    case JoinType.Right:
                        input_left, input_right = output, labels
                    case _:
                        raise "unreachable"

                output = DslPlan.HStack(
                    input=DslPlan.Join(
                        input_left,
                        input_right,
                        left_on,
                        right_on,
                        options,
                        predicates=[],
                    ),
                    exprs=fill_null,
                    options=ProjectionOptions.default(),
                )
        return output

    def privacy_map(d_in: Bounds):  #
        bound = d_in.get_bound(group_by_id)

        # incorporate all information into optional bounds
        l0 = option_min(bound.num_groups, margin.max_groups);
        li = option_min(bound.per_group, margin.max_length);
        l1 = d_in.get_bound(HashSet.new()).per_group; #

        # reduce optional bounds to concrete bounds
        if l0 is not None and l1 is not None and li is not None:
```

```python
            pass
        elif l1 is not None:
            l0 = l0 or l1 #
            li = li or l1 #
        elif l0 is not None and li is not None:
            l1 = l0.inf_mul(li) #
        else: #
            raise f"num_groups ({l0}), total contributions ({l1}), and per_group ({li}) are not sufficiently well-defined."

        d_out = f_privacy_map.eval((l0, l1, li))

        if margin.invariant is not None or is_join:  #
            pass
        elif threshold_info is not None:  #
            _, noise, threshold_value, _ = threshold_info
            if li >= threshold_value:
                raise f"Threshold must be greater than {li}."

            d_instability = threshold_value.neg_inf_sub(li)
            delta_single = integrate_discrete_noise_tail(
                noise.distribution, noise.scale, d_instability
            )
            delta_joint = (1).inf_sub(
                (1).neg_inf_sub(delta_single).neg_inf_powi(IBig.from_(l0))
            )
            d_out = MO.add_delta(d_out, delta_joint)
        else:
            raise "the key-set is sensitive"

        return d_out

    return t_prior >> Measurement.new(
        middle_domain,
        function,
        middle_metric,
        output_measure,
        privacy_map,
    )
```

Postconditions

Theorem 1.1. For every setting of the input parameters (`input_domain`, `input_metric`, `output_measure`, `plan`, `global_scale`, `threshold`, `MI`, `MO`) to `make_private_group_by` such that the given preconditions hold, `make_private_group_by` raises an exception (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 elements x, x' in `input_domain`, `function`(x) returns an error if and only if `function`(x') returns an error.

2. (Privacy 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_measure`, if x, x' are `d_in`-close under `input_metric`, `privacy_map`(`d_in`) does not raise an exception, and `privacy_map`(`d_in`) $\leq$ `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 55)
   (a) reconcile information about the join (line 58)
   (b) reconcile information about the threshold (line 67)
   (c) update key sanitizer (line 80)
4. build final measurement (line 98)
   (a) construct function (line 99)
   (b) construct privacy map (line 140)

`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).

Line 40 starts the process to prepare 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 grouping columns `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`, invokation 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 joining

An upper bound on the total number of partitions can be statically derived via the length of the grouping keys. Line 62 retrieves this information from the key-set and assigns it to the margin. `is_join` indicates that key sanitization will occur via a join.

2.3.2 Reconcile information when filtering

Line 67 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 68).
- Otherwise, if the key sanitizer contains filtering criteria (line 70), then by the postcondition of `find_len_expr`, filtering on `name` can be used to satisfy δ-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 74), 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 77 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 67 is intentionally written to ignore the constructor threshold when a suitable filtering threshold is already detected in the plan, to avoid overwriting/changing it.

2.3.3 Update key sanitizer

We now update `key_sanitization` starting from line 80:

- When filtering (line 81), `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 84. The first case preserves any additional filtering criteria that was already present in the plan, but not used for key release.
- When joining (line 89) 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 99 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 140 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 152.
- L^∞: 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 153.
- 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 146, but can also be bounded by the product of the L^0 and L^∞ bounds on line 155.

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 161 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 163. 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 $\Pr[E] \geq 1 - \delta$.

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

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 $\delta = 1 - (1 - \texttt{delta\_single})^{\Delta_0}$. 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.