fn match_truncations

Michael Shoemate

December 6, 2025

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

Proves soundness of `match_truncations` in `mod.rs` at commit f5bb719 (outdated[1]).

1 Hoare Triple

Compiler Verified

Types matching pseudocode.

Precondition

None

Function

```
def match_truncations(
    plan: DslPlan, identifier: Expr
) -> tuple[DslPlan, Vec[Truncation], Vec[Bound]]:
    truncations = []
    bounds = []

    allowed_keys = match_group_by_truncation(plan, identifier) #
    if allowed_keys:
        input, truncate, new_bound = allowed_keys
        plan = input #
        truncations.append(truncate)
        bounds.append(new_bound) #
        allowed_keys = new_bound.by #

    # match until not a filter truncation
    while isinstance(plan, Truncation.Filter): #
        input, predicate = plan.input, plan.predicate
        new_bounds = match_truncation_predicate(predicate, identifier) #
        if new_bounds is None:
            break

        # When filter truncation is behind a groupby truncation,
        # if the groupby group keys don't cover the filter truncation keys,
        # then the groupby aggs can overwrite the filter truncation keys,
        # invalidating the filter truncation bounds.
        if allowed_keys is not None: #
```

[1] See new changes with `git diff f5bb719..f969979 rust/src/transformations/make_stable_lazyframe/truncate/matching/mod.rs`

```
27              for bound in new_bounds:
28                  if not bound.by.is_subset(allowed_keys):
29                      raise f"Filter truncation keys ({bound.by}) must be a subset of groupby
    truncation keys ({allowed_keys})."
30
31          plan = input #
32          truncations.append(Truncation.Filter(predicate))
33          bounds.extend(new_bounds) #
34
35      # just for better error messages, no privacy implications
36      if match_group_by_truncation(plan, identifier) is not None: #
37          raise Exception("groupby truncation must be the last truncation in the plan.")
38
39      # since the parse descends to the source,
40      # truncations and bounds are in reverse order
41      truncations.reverse() #
42      bounds.reverse() #
43
44      return plan, truncations, bounds
```

Postcondition

Theorem 1.1 (Postcondition)**.** For any choice of LazyFrame plan, returns the plan with the truncations removed, the truncations that were removed in the order they are applied, and per-id bounds on row and/or group contributions.

Proof. The algorithm maintains three invariants:

- `input` is the LazyFrame plan with truncations removed
- `truncations` is a list of truncations in reverse-order
- `bounds` is a list of per-id bounds on row and/or group contributions

In order to ensure that all per-id bounds remain valid after successive truncations, the group by truncation may only be applied last in the truncation pipeline, as the group by truncation rewrites all columns in the data, and potentially overwrites user identifiers.

Since parsing the query plan happens in reverse order, the algorithm starts by attempting to parse a group by truncation on line 7. By the postcondition of `match_group_by_truncation`, if the group by truncation is present, it will be returned as a tuple of `input`, (the execution plan with the group by truncation removed), `truncation`, (the group by truncation) and `per_id_bounds` (the per-id bounds on row contributions). The state of the algorithm is then updated on lines 10-12, maintaining the invariants on `input`, `truncations` and `bounds`.

Another limitation of the group by truncation is that bounds on row contributions when grouped by a given set of columns are no longer valid if those columns are changed in the group by truncation. Therefore, `allowed_keys` on line 13 contains columns that are preserved through the group by truncation, by virtue of being part of the grouping columns. This limitation does not hinder expected use-cases, but is necessary to ensure that per-id bounds on contributions remain valid after the group by truncation.

If a group by truncation is not present, no update is made to the state.

The algorithm then attempts to repeatedly parse filter truncations on line 16. By the postcondition of `match_truncation_predicate` on line 18, the return is a list of per-id contribution bounds if the predicate consists solely of truncations, otherwise none. This ensures that the algorithm rejects predicates that contain conditions that are not truncations.

Line 26 checks that the truncation predicate is valid, by ensuring that the truncation predicate is a subset of the allowed keys. The algorithm then updates the state on lines 31-33, maintaining the invariants on `input`, `truncations` and `bounds`. Finally, since the descent through the query plan is in reverse order, line 41 ensures that the truncation order is correct. Neither the check on line 36 nor the reversal on line 42 are necessary to match the postcondition, but both are included to improve usability.

Since the invariants on `input`, `truncations` and `bounds` are maintained, and the algorithm only matches through truncations, the postcondition is satisfied. □