# Common Sub-Expression Optimization in Presto

Rongrong (rongrong@fb.com)

Facebook

# The Problem

```
Fragment 0 [SINGLE]
    Output layout: [expr, expr_0, expr_1, expr_2, expr_3, expr_4]
    Output partitioning: SINGLE []
    Stage Execution Strategy: UNGROUPED_EXECUTION
    - Output[a, b, c, d, e, f] => [expr:row(adcampaign_old_promoted_object_id bigint, adcampaign_promoted_object_id bigint), expr_0:row(uninvoiced_currency varchar, uninvoiced_usd_amount integer), expr_1:row(adgroup_id bigint), expr_2:row(behavior varchar), expr_3:row(credit_card_fraud_check_source
varchar, credit_card_unique_id varchar), expr_4:row(bm_id bigint, business_ad_account_action varchar)]
            Estimates: {rows: 1000 (298.83kB), cpu: 652059180.00, memory: 0.00, network: 162938295.00}
            a := expr
            b := expr_0
            c := expr_1
            d := expr_2
            e := expr_3
            f := expr_4
        - Project[] => [expr:row(adcampaign_old_promoted_object_id bigint, adcampaign_promoted_object_id bigint), expr_0:row(uninvoiced_currency varchar, uninvoiced_usd_amount integer), expr_1:row(adgroup_id bigint), expr_2:row(behavior varchar), expr_3:row(credit_card_fraud_check_source varchar,
credit_card_unique_id varchar), expr_4:row(bm_id bigint, business_ad_account_action varchar)]
                Estimates: {rows: 1000 (298.83kB), cpu: 652059180.00, memory: 0.00, network: 162938295.00}
                expr := TRY_CAST(json_parse(features))
                expr_0 := TRY_CAST(json_parse(features))
                expr_1 := TRY_CAST(json_parse(features))
                expr_2 := TRY_CAST(json_parse(features))
                expr_3 := TRY_CAST(json_parse(features))
                expr_4 := TRY_CAST(json_parse(features))
            - LocalExchange[ROUND_ROBIN] () => [features:varchar]
                    Estimates: {rows: 1000 (155.39MB), cpu: 651753180.00, memory: 0.00, network: 162938295.00}
                - Limit[10000] => [features:varchar]
                        Estimates: {rows: 1000 (155.39MB), cpu: 488814885.00, memory: 0.00, network: 162938295.00}
                    - LocalExchange[SINGLE] () => [features:varchar]
                            Estimates: {rows: 1000 (155.39MB), cpu: 325876590.00, memory: 0.00, network: 162938295.00}
                        - RemoteSource[1] => [features:varchar]

Fragment 1 [SOURCE]
    Output layout: [features]
    Output partitioning: SINGLE []
    Stage Execution Strategy: UNGROUPED_EXECUTION
    - LimitPartial[10000] => [features:varchar]
            Estimates: {rows: 1000 (155.39MB), cpu: 325876590.00, memory: 0.00, network: 0.00}
        - TableScan[TableHandle {connectorId='prism', connectorHandle='HiveTableHandle{schemaName=ad_metrics, tableName=demo_json_parse, analyzePartitionValues=Optional.empty}', layout='Optional[ad_metrics.demo_json_parse{}]'}, grouped = false] => [features:varchar]
                Estimates: {rows: 1000 (155.39MB), cpu: 162938295.00, memory: 0.00, network: 0.00}
                LAYOUT: ad_metrics.demo_json_parse{}
                features := features:string:0:REGULAR
                ts:string:-14:PARTITION_KEY
                    :: [[2020-06-02+03:00:99], [2020-06-02+04:00:99], [2020-06-02+05:00:99], [2020-06-02+06:00:99]]
                ds:string:-13:PARTITION_KEY
                    :: [[2020-06-01]]
```

The same expression is evaluated 6 times

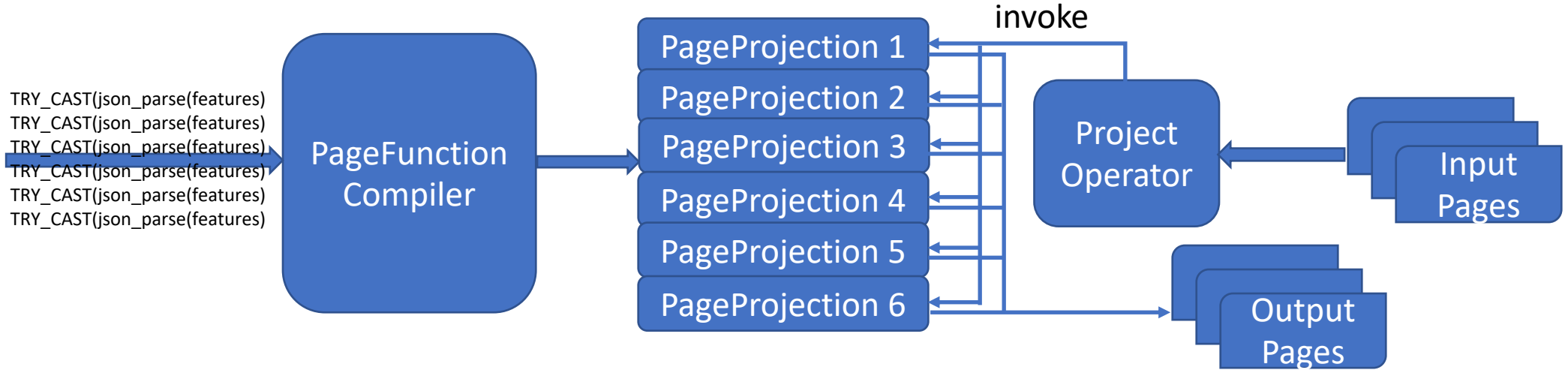
# The solution

- Run only once

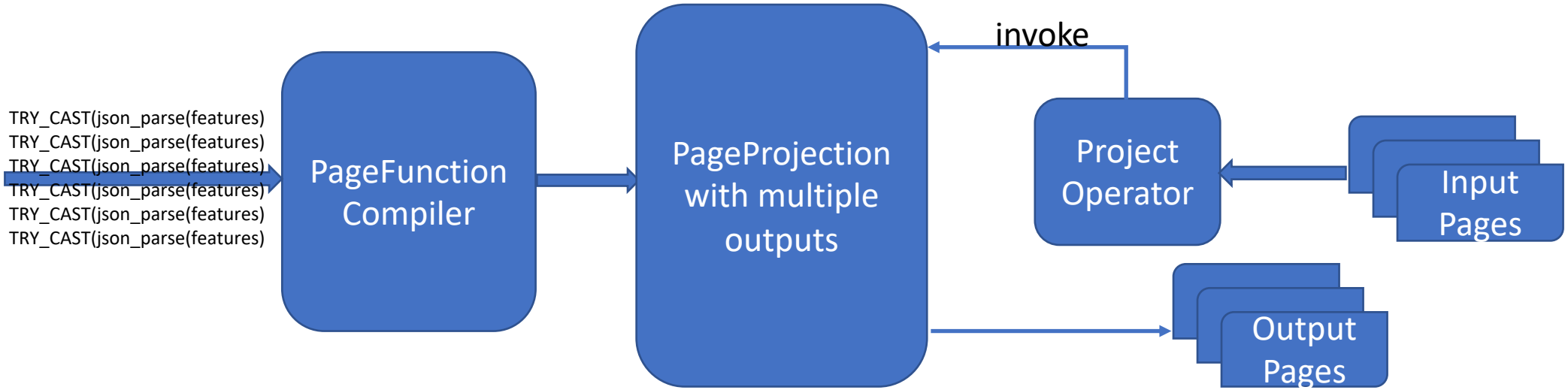

- How it works?

# How Projection Works in Presto (without CSE)

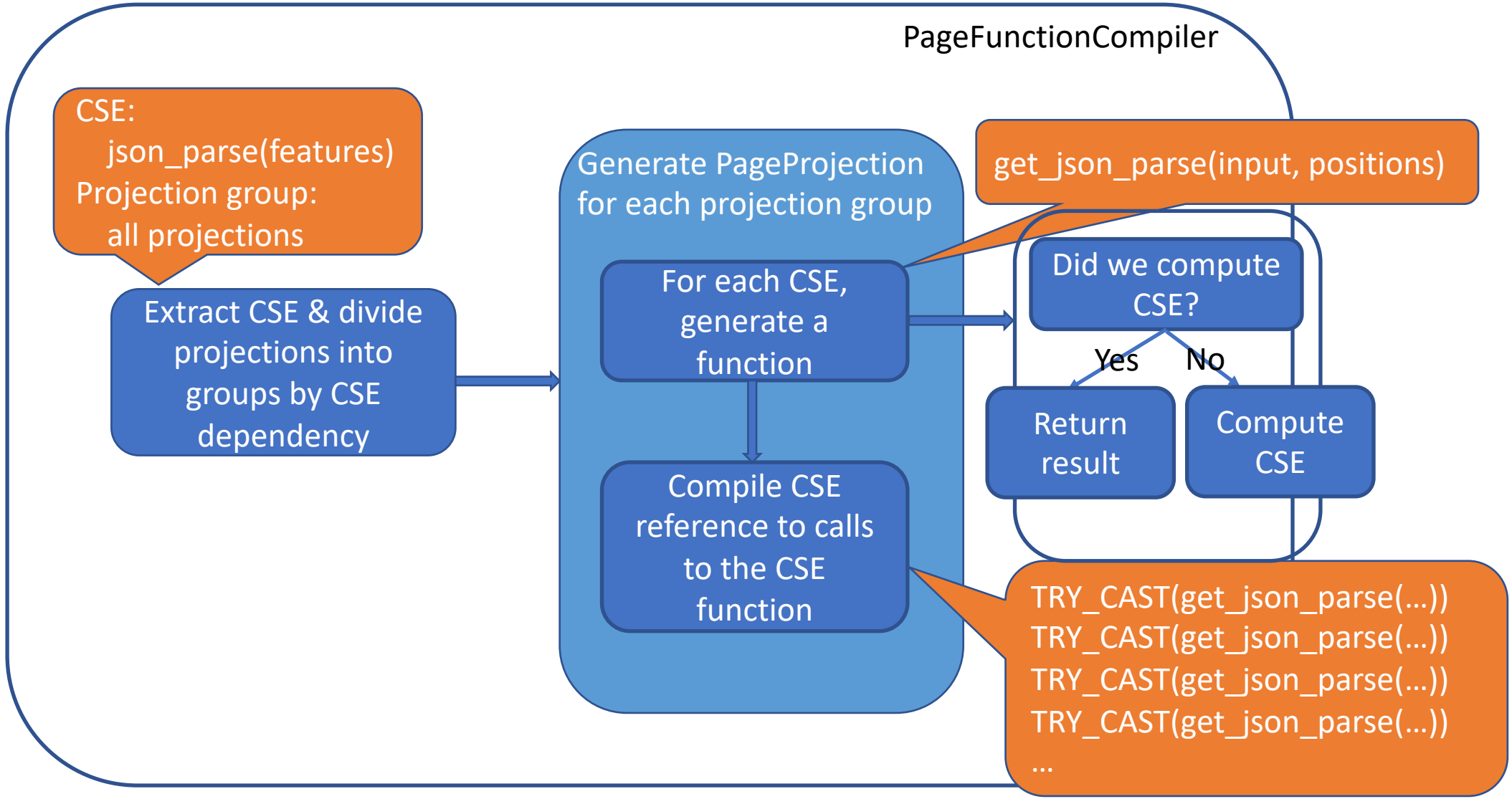
- Presto generates java bytecode directly for expression evaluation
- Generate PageProjection class for each assignment
  - For the example query, we will generate 6 separate PageProject class, each processing one assignment
- ProjectOperator goes through all assignments, invoke each generated PageProjection.project on a given page



# How CSE works



# Inside PageFunctionCompiler



# Another Example

```
SELECT x + y + z, x + y * z, (x + y + z) * 2, cast(x + y + z AS VARCHAR), (x + y + z) * 2 * z FROM (VALUES (1, 2, 4), (3, 5, 7), (2, 4, 5)) t(x, y, z);
```

```
- Project□ => [expr_10:integer, expr_11:integer, expr_12:integer, expr_13:varchar, expr_14:integer]
        Estimates: {rows: 3 (225B), cpu: 270.00, memory: 0.00, network: 0.00}
        expr_10 := ((field) + (field_0)) + (field_1)
        expr_11 := (field) + ((field_0) * (field_1))
        expr_12 := (((field) + (field_0)) + (field_1)) * (INTEGER 2)
        expr_13 := CAST(((field) + (field_0)) + (field_1) AS varchar)
        expr_14 := ((((field) + (field_0)) + (field_1)) * (INTEGER 2)) * (field_1)
```

- Generate projections
  - Group 1: expr_11 = x + y * z
  - Group 1:
    - cse1 = x + y + z
    - cse2 = cse1 * 2
    - expr_10 = cse1
    - expr_12 = cse2
    - expr_13 = CAST(cse1 AS VARCHAR)
    - Expr_14 = cse2 * z

Will not generate CSE for x + y

# Performance

- Microbenchmark (CommonSubExpessionBenchmark)

10% + performance improvements for simple operations like x + y

| Benchmark | (dictionaryBlocks) | (functionType) | (optimizeCommonSubExpression) | Mode | Cnt | Score | Error | Units |
|---|---|---|---|---|---|---|---|---|
| | true | json | true | avgt | 20 | 9633.238 ± | 395.197 | ns/op |
| | true | json | false | avgt | 20 | 10515.592 ± | 409.112 | ns/op |
| | true | bigint | true | avgt | 20 | 4345.824 ± | 224.361 | ns/op |
| | true | bigint | false | avgt | 20 | 4989.348 ± | 178.232 | ns/op |
| | true | varchar | true | avgt | 20 | 4335.683 ± | 142.163 | ns/op |
| | true | varchar | false | avgt | 20 | 5079.818 ± | 226.231 | ns/op |
| CommonSubExpressionBenchmark.compute | false | json | true | avgt | 20 | 1306270.790 ± | 45640.515 | ns/op |
| CommonSubExpressionBenchmark.compute | false | json | false | avgt | 20 | 1757459.750 ± | 89664.951 | ns/op |
| CommonSubExpressionBenchmark.compute | false | bigint | true | avgt | 20 | 11733.285 ± | 621.099 | ns/op |
| CommonSubExpressionBenchmark.compute | false | bigint | false | avgt | 20 | 12266.889 ± | 408.858 | ns/op |
| CommonSubExpressionBenchmark.compute | false | varchar | true | avgt | 20 | 101828.239 ± | 2372.572 | ns/op |
| CommonSubExpressionBenchmark.compute | false | varchar | false | avgt | 20 | 133340.871 ± | 4959.539 | ns/op |

- 3x performance for expensive expressions like JSON_PARSE, REGEX

# Downside?

- Larger bytecode class (can potentially trigger compiler failure)
  - Mostly mitigated by partitioning projections into groups and having an upper limit of 10 projections / group
- Larger function (causing wall time regression)
  - Mostly mitigated by having an upper limit of 10 projections / group
- Still have issue?
  - Turn it off with optimize_common_sub_expressions = false

# Common Sub Expressions in Filter

- Filter will always have at most one expression
- Same approach applies to Filter
  - CSE within the filter expression
  - Seems to be common especially with generated queries

# What’s next

- Support common sub-expressions in CursorProcessorCompiler
- Bugs? Regressions? Let us know!