Calcite相关日志分析

1. `14:57:20 [http-nio-9080-exec-2] DEBUG org.apache.calcite.sql.parser 890 - Reduced DAYOFWEEK( imp_date ) - 2`

- **含义**: 这行日志记录了SQL解析过程中对 `DAYOFWEEK(imp_date) - 2` 表达式的简化。Calcite解析器正在处理SQL表达式，并减少了其中的复杂性。

2. `14:57:22 [http-nio-9080-exec-2] DEBUG org.apache.calcite.sql2rel 628 - Plan after converting SqlNode to RelNode`

- **含义**: 这条日志表明，Calcite已经完成了从 `SqlNode` （SQL解析树）到 `RelNode` （关系表达式树）的转换，并显示了转换后的逻辑执行计划。
- **计划细节**:
  - `LogicalProject(sys_imp_date=[$0], pv=[$2])`: 这是一个投影操作，选择了 `sys_imp_date` 和 `pv` 两列。
  - `LogicalAggregate(group=[{0, 1}], s2_pv_uv_statis_pv=[SUM($2)])`: 这是一个聚合操作，按 `sys_imp_date` 和 `imp_date` 分组，并计算 `s2_pv_uv_statis_pv` 的总和。
  - `LogicalTableScan(table=[[s2_pv_uv_statis]])`: 这表示对 `s2_pv_uv_statis` 表的扫描。

3. `14:57:22 [http-nio-9080-exec-2] DEBUG c.t.s.h.c.t.c.sql.node.SemanticNode 423 - RelNode optimize SELECT imp_date AS sys_imp_date, SUM(1) AS pv FROM s2_pv_uv_statis GROUP BY imp_date, imp_date`

- **含义**: 这个日志记录了经过优化的查询计划，显示了SQL查询优化后的形式。这里展示的是一个聚合查询，计算 `s2_pv_uv_statis` 表中 `imp_date` 字段的总和，并将其作为 `sys_imp_date` 返回。
- **优化后的SQL**:

```
SELECT imp_date AS sys_imp_date, SUM(1) AS pv
FROM s2_pv_uv_statis
```

```
GROUP BY imp_date, imp_date
```

4. **`14:57:22 [http-nio-9080-exec-2] DEBUG o.a.c.p.A.rule_execution_summary 300 - Rule Attempts Info for HepPlanner`**

- **含义**: 这条日志显示了HepPlanner（Calcite的规划器之一）在优化过程中尝试应用的规则信息。通常情况下，HepPlanner会应用一系列规则来优化查询。

5. **`14:57:22 [http-nio-9080-exec-2] DEBUG o.a.c.p.A.rule_execution_summary 301 - Rules Attempts Time (us) * Total 0 0`**

- **含义**: 这个日志记录显示，没有任何规则被应用或尝试优化（Attempts为0，时间也为0微秒）。这可能表示该特定步骤或优化阶段没有执行任何规则应用。

6. **`14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.RelOptPlanner 380 - For final plan, using rel#18:LogicalProject.(input=HepRelVertex#17,inputs=0,exprs=[$2])`**

- **含义**: 这是日志显示在最终计划中使用的一个`LogicalProject`操作，指定了从HepPlanner（Calcite优化器的一部分）产生的一个顶点作为输入，并应用了一个投影操作。

7. **`14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.RelOptPlanner 380 - For final plan, using rel#16:LogicalAggregate.(input=HepRelVertex#15,group={0, 1},s2_pv_uv_statis_pv=SUM($2))`**

- **含义**: 这行日志记录了另一个优化后的逻辑计划，显示了一个`LogicalAggregate`操作，它根据分组字段`{0, 1}`（通常对应于`sys_imp_date`和`imp_date`）进行聚合，并计算`s2_pv_uv_statis_pv`的总和。

8. **`14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.RelOptPlanner 380 - For final plan, using rel#14:LogicalProject.(input=HepRelVertex#13,exprs=[$4, $4, 1])`**

- **含义**: 这是另一个`LogicalProject`操作，在最终的计划中投影了表达式`[$4, $4, 1]`。这里的`$4`可能是某个字段的索引值，`1`是常量值。

9. `14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.RelOptPlanner 380 - For final plan, using rel#1:LogicalTableScan.(table=[s2_pv_uv_statis])`

- **含义**: 最后，日志记录了最终计划中使用的 `LogicalTableScan` 操作，这表示将扫描 `s2_pv_uv_statis` 表的数据作为执行计划的一部分。

总结

录了从SQL语句解析、优化到最终执行计划的生成过程。Apache Calcite先是简化了SQL表达式，然后将其转换为关系表达式（RelNode），再通过应用各种优化规则生成了最终的逻辑执行计划。这个过程展示了如何有效地从SQL到物理执行计划的转换，并通过优化器应用了多种优化策略。

10.

```
14:57:22 [http-nio-9080-exec-2] DEBUG org.apache.calcite.sql.parser 890 - Reduced `sys_imp_date` >= '2024-07-31' AND `sys_imp_date` <= '2024-08-06'
```

- **解释**:
  - 表示 Calcite SQL 解析器已经解析并简化了条件表达式，该表达式检查 `sys_imp_date` 字段是否在 `'2024-07-31'` 和 `'2024-08-06'` 之间。这是 SQL 解析和优化过程中常见的一步，确保表达式在进一步处理和执行之前尽可能简洁和高效。

11.

```
14:57:22 [http-nio-9080-exec-2] DEBUG org.apache.calcite.sql2rel 628 - Plan after converting SqlNode to RelNode
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)])
  LogicalProject(pv=[$1])
    LogicalFilter(condition=[AND(>=($0, _UTF-8'2024-07-31'), <=($0, _UTF-8'2024-08-06'))])
      LogicalProject(sys_imp_date=[$0], pv=[$1])
        LogicalAggregate(group=[{0}], pv=[SUM($1)])
          LogicalProject(imp_date=[$4], $f1=[1])
            LogicalTableScan(table=[[s2_pv_uv_statis]])
```

- **解释**:
  - 这一行展示了 SQL 查询在转换为逻辑查询计划 (RelNode) 后的样子。Calcite 使用 Relational Algebra（关系代数）的表示法来描述查询操作的逻辑步骤。
  - **具体逻辑步骤**:
    1. **LogicalAggregate**:
       - 这是一个聚合操作（如 SUM、COUNT 等）。
       - `group=[{}]` 表示没有分组字段（即全局聚合）。
       - `EXPR$0=[SUM($0)]` 表示对 `$0` 列进行 SUM 聚合运算。
    2. **LogicalProject**:
       - 表示列投影操作，选择和重命名特定列。
       - `pv=[$1]` 表示选择第二列并将其命名为 `pv`。
    3. **LogicalFilter**:
       - 表示过滤操作（类似 SQL 的 WHERE 子句）。
       - `condition=[AND(>=($0, _UTF-8'2024-07-31'), <=($0, _UTF-8'2024-08-06'))]` 表示对第一个列（$0，也就是 `sys_imp_date`）应用条件过滤，检查其是否在 `2024-07-31` 和 `2024-08-06` 之间。
    4. **LogicalProject**:
       - 再次进行列投影，选择 `sys_imp_date` 和 `pv` 列。
    5. **LogicalAggregate**:
       - 对第一个列（$0，也就是 `sys_imp_date`）进行分组，对第二个列（$1，也就是 `pv`）进行 SUM 聚合。
    6. **LogicalProject**:
       - 再次进行列投影，选择 `imp_date` 并创建一个常量列 `$f1`，值为 1。
    7. **LogicalTableScan**:
       - 表示从物理表 `s2_pv_uv_statis` 中扫描数据，这是逻辑计划中最底层的操作，用于从实际的数据表中读取数据。

12.

```
14:57:22 [http-nio-9080-exec-2] DEBUG org.apache.calcite.sq
l2rel 628 - Plan after converting SqlNode to RelNode
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)])
  LogicalProject(pv=[$1])
    LogicalFilter(condition=[AND(>=($0, _UTF-8'2024-07-3
1'), <=($0, _UTF-8'2024-08-06'))])
      LogicalProject(sys_imp_date=[$0], pv=[$1])
        LogicalAggregate(group=[{0}], pv=[SUM($1)])
          LogicalProject(imp_date=[$4], $f1=[1])
            LogicalTableScan(table=[[s2_pv_uv_statis]])
```

- **解释**:
  - 这行与前一行的内容完全相同，Calcite 再次展示了逻辑查询计划的转换结果。这可能是在日志输出时的重复展示，确保在优化和转换过程的不同阶段查看该计划。

13. semanticNode 代码解析

`com.tencent.supersonic.headless.core.translator.calcite.sql.node` 包，主要定义了 `SemanticNode` 抽象类。这类代码使用了 Apache Calcite 进行 SQL 解析、优化和转换，是整个 SQL 处理链条中的一部分。下面我将逐步分析代码的各个部分及其作用。

类和包的结构

- **包名:** `com.tencent.supersonic.headless.core.translator.calcite.sql.node`
  - 该包通常用于处理 SQL 语句的解析和转换，将 SQL 语句转化为 Calcite 的内部表达方式（如 RelNode）并执行优化。
- **类名:** `SemanticNode`
  - 这是一个抽象类，提供了多个静态方法来解析、处理、优化和转换 SQL 语句。

核心字段

```
public static Set<SqlKind> AGGREGATION_KIND = new HashSet<>
();
public static Set<String> AGGREGATION_FUNC = new HashSet<>
();
```

```
public static List<String> groupHints = new ArrayList<>(Arr
ays.asList("1", "2", "3", "4", "5", "6", "7", "8", "9"));
```

- **AGGREGATION_KIND**:
  - 这个集合存储了所有支持的聚合操作（如 AVG、COUNT、SUM、MAX、MIN）。`SqlKind` 是 Calcite 用来表示 SQL 语句中不同操作类型的枚举。
- **AGGREGATION_FUNC**:
  - 该集合存储了聚合函数的名称字符串（如 "sum", "count", "max", "avg", "min"），用于匹配和识别 SQL 中的聚合函数。
- **groupHints**:
  - 这是一个存储数字字符串的列表，用于处理 SQL 中的分组提示（Group hints），这些可能是对 SQL 中分组操作的特定索引或标识。

核心方法

1. **parse** 方法

```
public static SqlNode parse(String expression, SqlValidator
Scope scope, EngineType engineType) throws Exception {
    SqlValidatorWithHints sqlValidatorWithHints = Configura
tion.getSqlValidatorWithHints(
            scope.getValidator().getCatalogReader().getRoot
Schema(), engineType);
    if (Configuration.getSqlAdvisor(sqlValidatorWithHints,
engineType).getReservedAndKeyWords()
            .contains(expression.toUpperCase())) {
        expression = String.format("`%s`", expression);
    }
    SqlParser sqlParser = SqlParser.create(expression, Conf
iguration.getParserConfig(engineType));
    SqlNode sqlNode = sqlParser.parseExpression();
    scope.validateExpr(sqlNode);
    return sqlNode;
}
```

- **功能**:
  - 该方法解析输入的 SQL 表达式，将其转换为 Calcite 的 `SqlNode` 对象。

- 首先，通过 `SqlValidatorWithHints` 验证表达式是否包含保留字，如果是，则将表达式包裹在反引号中。
- 然后使用 Calcite 的 `SqlParser` 进行 SQL 解析，将其转换为 `SqlNode`。
- 最后，对解析后的 `SqlNode` 进行验证（如语法检查）。

1. **buildAs** 方法

```
public static SqlNode buildAs(String asName, SqlNode sqlNod
e) throws Exception {
    SqlAsOperator sqlAsOperator = new SqlAsOperator();
    SqlIdentifier sqlIdentifier = new SqlIdentifier(asName,
SqlParserPos.ZERO);
    return new SqlBasicCall(sqlAsOperator, new ArrayList<>
(Arrays.asList(sqlNode, sqlIdentifier)),
            SqlParserPos.ZERO);
}
```

- **功能**:
  - 构建一个 SQL 的 `AS` 操作（用于给表达式或字段起别名）。
  - `sqlNode` 是需要重命名的 SQL 表达式或字段，而 `asName` 是新名称。
  - 返回一个包含 `AS` 操作的 `SqlNode`。

1. **getSql** 方法

```
public static String getSql(SqlNode sqlNode, EngineType eng
ineType) {
    UnaryOperator<SqlWriterConfig> sqlWriterConfigUnaryOper
ator = (c) -> getSqlWriterConfig(engineType);
    return sqlNode.toSqlString(sqlWriterConfigUnaryOperato
r).getSql();
}
```

- **功能**:
  - 将 `SqlNode` 转换为 SQL 字符串表示。
  - 通过 `SqlWriterConfig` 配置 SQL 的输出格式，生成适合指定 `engineType`（引擎类型）的 SQL 语句。

1. **isNumeric** 方法

```
public static boolean isNumeric(String expr) {
    return StringUtils.isNumeric(expr);
}
```

- **功能**:
  - 检查给定的表达式 `expr` 是否为数值。该方法利用 Apache Commons Lang3 库中的 `StringUtils.isNumeric` 方法。

1. **expand** 方法

```
public static List<SqlNode> expand(SqlNode sqlNode, SqlVali
datorScope scope) throws Exception {
    if (!isIdentifier(sqlNode)) {
        List<SqlNode> sqlNodeList = new ArrayList<>();
        expand(sqlNode, sqlNodeList);
        return sqlNodeList;
    }
    return new ArrayList<>(Arrays.asList(sqlNode));
}
```

- **功能**:
  - 递归地展开 `SqlNode`，将复杂的表达式分解成更小的组成部分（如字段标识符、操作符等）。
  - 该方法用于处理和拆解复杂的 SQL 表达式，使其便于进一步的处理和分析。

1. **optimize** 方法

```
public static SqlNode optimize(SqlValidatorScope scope, Sem
anticSchema schema, SqlNode sqlNode,
                               EngineType engineType) {
    try {
        HepProgramBuilder hepProgramBuilder = new HepProgra
mBuilder();
        SemanticSqlDialect sqlDialect = SqlDialectFactory.g
etSqlDialect(engineType);
        hepProgramBuilder.addRuleInstance(new FilterToGroup
```

```
ScanRule(FilterToGroupScanRule.DEFAULT, schema));
        RelOptPlanner relOptPlanner = new HepPlanner(hepPro
gramBuilder.build());
        RelToSqlConverter converter = new RelToSqlConverter
(sqlDialect);
        SqlValidator sqlValidator = Configuration.getSqlVal
idator(
                scope.getValidator().getCatalogReader().get
RootSchema(), engineType);
        SqlToRelConverter sqlToRelConverter = Configuratio
n.getSqlToRelConverter(scope, sqlValidator,
                relOptPlanner, engineType);
        RelNode sqlRel = sqlToRelConverter.convertQuery(
                sqlValidator.validate(sqlNode), false, tru
e).rel;
        log.debug("RelNode optimize {}",
                SemanticNode.getSql(converter.visitRoot(sql
Rel).asStatement(), engineType));
        relOptPlanner.setRoot(sqlRel);
        RelNode relNode = relOptPlanner.findBestExp();
        return converter.visitRoot(relNode).asStatement();
    } catch (Exception e) {
        log.error("optimize error {}", e);
    }
    return null;
}
```

- **功能**:
  - 该方法对 `SqlNode` 进行优化，将其转换为 Calcite 的 `RelNode` 形式，并通过优化器（`HepPlanner`）进行优化处理。
  - 之后将优化后的 `RelNode` 再转换为 SQL 语句（`SqlNode`），并返回。
  - 这个过程包含 SQL 验证、转换为关系代数形式（RelNode）、应用优化规则等步骤。

14. `14:57:22 [http-nio-9080-exec-2] DEBUG c.t.s.h.c.t.c.sql.node.SemanticNode 423 - RelNode optimize SELECT SUM(`pv`)
FROM
(SELECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
FROM
s2_pv_uv_statis
GROUP BY `imp_date`) AS `t1`
WHERE `sys_imp_date` >= '2024-07-31' AND `sys_imp_date` <= '2024-08-06'`

结合代码分析

1. **SQL 查询结构**:
   - 最外层的查询语句是 `SELECT SUM(pv) FROM ... WHERE sys_imp_date >= '2024-07-31' AND sys_imp_date <= '2024-08-06'`。
   - 内层子查询为 `SELECT imp_date AS sys_imp_date, SUM(1) AS pv FROM s2_pv_uv_statis GROUP BY imp_date`。
2. **子查询**:
   - 内层子查询从 `s2_pv_uv_statis` 表中选取数据，将 `imp_date` 字段重命名为 `sys_imp_date`。
   - 同时，对于每个 `imp_date`，计算其对应的 `pv`（访问次数）的总和。这里的 `SUM(1)` 表示对每一行的计数，通常是计算某个条件下的行数总和。
3. **外层查询**:
   - 外层查询对内层查询的结果进行汇总，计算出 `pv` 字段（即访问次数）的总和。
   - 并对 `sys_imp_date` 进行过滤，只选取在 `'2024-07-31'` 和 `'2024-08-06'` 之间的日期范围的数据。

Calcite 解析和优化过程

在 `SemanticNode` 类中，有一个核心的 `optimize` 方法负责优化 SQL 查询。

1. **初始解析**:
   - 通过 `SqlParser` 解析原始的 SQL 表达式，生成 `SqlNode`。

2. **SQL 验证**:
    - 使用 `SqlValidator` 对 `SqlNode` 进行验证，检查语法和语义上的合法性。
3. **转换为 RelNode**:
    - `SqlToRelConverter` 将经过验证的 `SqlNode` 转换为 `RelNode`，这一步将 SQL 语句转化为 Calcite 内部使用的关系代数树。
4. **优化**:
    - `HepPlanner` 使用一组优化规则（如合并过滤条件、推导下推、消除冗余等）对生成的 `RelNode` 进行优化。这个过程中可能会对查询进行重写，生成更高效的执行计划。
5. **生成最终 SQL**:
    - 经过优化后的 `RelNode` 再次转换回 `SqlNode`，最后由 `RelToSqlConverter` 生成最终的 SQL 字符串。

总结

这条日志展示了 Calcite 在优化 SQL 查询后的结果。具体来说，Calcite 通过以下步骤生成了最终 SQL：

1. 内层子查询通过聚合和分组将原始数据表 `s2_pv_uv_statis` 中的数据按日期分组，并计算每个日期对应的 `pv`。
2. 外层查询汇总了 `pv` 的总和，并通过日期范围过滤符合条件的数据。
3. 这种嵌套查询的形式可能是经过 Calcite 的优化规则决定的，以实现更高效的数据查询和处理。

15.

日志解析

```
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.a.c.p.A.rule_executio
14:57:22 [http-nio-9080-exec-2] DEBUG o.a.c.p.A.rule_executio
Rules
FilterToGroupScanRule
* Total
```

日志内容解析

1. **Rule Application (`RelOptPlanner` 规则应用)**:
   - `RelOptPlanner` 是 Calcite 中用于优化关系代数表达式的规划器。它通过应用一系列规则来优化查询执行计划。
   - `call#0: Apply rule [FilterToGroupScanRule]` 这条日志表明，规划器正在应用 `FilterToGroupScanRule` 规则，并将它应用到了一组逻辑操作符上：
     - `LogicalFilter` (rel#49)
     - `LogicalProject` (rel#47)
     - `LogicalAggregate` (rel#45)
     - `LogicalProject` (rel#43)
2. **Rule Execution Summary (规则执行摘要)**:
   - `Rule Attempts Info for HepPlanner` 这一部分日志给出了一个摘要，显示了 HepPlanner 在应用优化规则时的尝试次数和所用时间。
   - `FilterToGroupScanRule` 规则在一次尝试中成功应用，并花费了 968 微秒（0.968 毫秒）的时间完成了这一操作。

`FilterToGroupScanRule` 规则

`FilterToGroupScanRule` 是 Calcite 中的一条优化规则，通常用于将 `Filter` 操作符下推到 `GroupBy` 之前的阶段，从而减少要处理的数据量。这种操作通常可以提高查询的效率，因为在数据聚合之前就已经将不必要的数据过滤掉了。

结合之前的查询优化

在之前的日志中，SQL 查询被转换为一个嵌套查询结构，并且在外层有一个过滤条件。`FilterToGroupScanRule` 识别到了可以在聚合之前应用过滤条件的机会，从而调整了逻辑计划中的操作符顺序，以实现更有效的执行计划。

- **Initial Plan**:
  - `LogicalProject` → `LogicalAggregate` → `LogicalFilter` → `LogicalProject`
- **Optimized Plan**:
  - `LogicalFilter` 被下推到 `LogicalAggregate` 之前，这样可以先过滤数据再进行聚合操作。

16.

日志分析

```
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
14:57:22 [http-nio-9080-exec-2] DEBUG o.apache.calcite.plan.R
```

1. `rel#53:LogicalAggregate`

- **操作**: 聚合操作（`SUM($0)`）。
- **输入**: `HepRelVertex#52`。
- **详细描述**: 这个操作符执行聚合操作，这里是 `SUM($0)`，表示对之前步骤中的某个列（即第一个列 `$0`）进行求和操作。这个操作没有 `group by`，意味着它在整个数据集上计算总和。

2. `rel#51:LogicalProject`

- **操作**: 投影操作。
- **输入**: `HepRelVertex#50`。
- **详细描述**: 该操作符从之前的结果中选择出特定的列（这里是 `$1` 列）。`LogicalProject` 用于选择特定的列或对列进行计算。

3. `rel#49:LogicalFilter`

- **操作**: 过滤操作。
- **输入**: `HepRelVertex#48`。
- **过滤条件**: `AND(>=($0, _UTF-8'2024-07-31'), <=($0, _UTF-8'2024-08-06'))`。
- **详细描述**: 这个操作符用于过滤数据，只保留满足指定条件的数据行。这里的条件是 `sys_imp_date` 的值在 `2024-07-31` 到 `2024-08-06` 之间。过滤操作通常用于减少后续操作所需处理的数据量。

4. `rel#47:LogicalProject`

- **操作**: 投影操作。
- **输入**: `HepRelVertex#46`。
- **详细描述**: 另一个投影操作符，从输入数据中选择某些特定列或进行列的计算。这里它将选择列 `exprs=[$1]`。

5. `rel#45:LogicalAggregate`

- **操作**: 聚合操作。
- **输入**: `HepRelVertex#44`。
- **分组依据**: `group={0}`，根据第0列进行分组。
- **聚合操作**: `pv=SUM($1)`。
- **详细描述**: 这个聚合操作符按照第 `0` 列进行分组，并对第 `1` 列的值进行求和操作，计算 `pv` 的总和。

6. `rel#43:LogicalProject`

- **操作**: 投影操作。
- **输入**: `HepRelVertex#42`。
- **详细描述**: 该操作选择第 `4` 列和常量 `1`，可能是为了创建一个新的列或者为后续操作做准备。

7. `rel#22:LogicalTableScan`

- **操作**: 表扫描。
- **表**: `s2_pv_uv_statis`。
- **详细描述**: 这是整个查询操作的起点，表示对物理表 `s2_pv_uv_statis` 进行扫描，读取所有相关数据。`LogicalTableScan` 通常是最底层的操作符，用于从数据库中读取数据。

总结

这段日志显示了 Calcite 在优化 SQL 查询时生成的最终逻辑执行计划。该计划首先通过 `LogicalTableScan` 读取数据，然后通过一系列 `LogicalProject`、`LogicalAggregate` 和 `LogicalFilter` 操作符对数据进行处理。最终的执行计划是通过优化器应用各种规则（例如之前提到的 `FilterToGroupScanRule`）生成的，以提高查询的执行效率。

17.

1. `simplifySql` 日志行

```
14:57:22 [http-nio-9080-exec-2] DEBUG c.t.s.h.c.t.c.Calcite
QueryParser 46 - simplifySql [SELECT SUM(`pv`)
FROM
(SELECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
FROM
s2_pv_uv_statis
GROUP BY `imp_date`) AS `t7`
WHERE `sys_imp_date` >= '2024-07-31' AND `sys_imp_date` <=
'2024-08-06']
```

- **作用**: 这一行展示了 `CalciteQueryParser` 在简化 SQL 查询时生成的中间 SQL 语句。
- **简化后的 SQL**:

  ```
  SELECT SUM(`pv`)
  FROM
  (SELECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
  FROM
  s2_pv_uv_statis
  GROUP BY `imp_date`) AS `t7`
  WHERE `sys_imp_date` >= '2024-07-31' AND `sys_imp_date`
  <= '2024-08-06'
  ```

- **分析**:
  - 该 SQL 查询先从 `s2_pv_uv_statis` 表中按 `imp_date` 分组，并计算 `SUM(1)` 作为 `pv`。`imp_date` 被别名为 `sys_imp_date`。
  - 然后，外层查询对计算得到的 `pv` 进行求和，同时通过 `sys_imp_date` 过滤数据，只保留时间范围在 `2024-07-31` 到 `2024-08-06` 之间的记录。

2. `before handleNoMetric` 日志行

```
14:57:22 [http-nio-9080-exec-2] DEBUG c.t.s.h.c.t.DetailQue
ryOptimizer 27 - before handleNoMetric, sql:with t_1 as (SE
LECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
FROM
```

```
s2_pv_uv_statis
GROUP BY `imp_date`, `imp_date`)
SELECT SUM(pv) FROM t_1 WHERE (sys_imp_date >= '2024-07-31'
AND sys_imp_date <= '2024-08-06') limit 1000
```

- **作用**: `DetailQueryOptimizer` 显示了在执行 `handleNoMetric` 优化前的 SQL 查询。
- **SQL 语句**:

```
with t_1 as (
    SELECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
    FROM s2_pv_uv_statis
    GROUP BY `imp_date`, `imp_date`
)
SELECT SUM(pv) FROM t_1 WHERE (sys_imp_date >= '2024-07-
31' AND sys_imp_date <= '2024-08-06') limit 1000
```

- **分析**:
  - 这段 SQL 使用了一个 `WITH` 子句，将子查询结果命名为 `t_1`。
  - 子查询内容与上面简化后的 SQL 基本相同，但这里 `GROUP BY` 中重复指定了 `imp_date`。
  - 最终查询的目的是对 `t_1` 中满足日期条件的 `pv` 进行求和，且限制结果集为 1000 行。

3. `after handleNoMetric` 日志行

```
14:57:22 [http-nio-9080-exec-2] DEBUG c.t.s.h.c.t.DetailQue
ryOptimizer 36 - after handleNoMetric, sql:with t_1 as (SEL
ECT `imp_date` AS `sys_imp_date`, SUM(1) AS `pv`
FROM
s2_pv_uv_statis
GROUP BY `imp_date`, `imp_date`)
SELECT SUM(pv) FROM t_1 WHERE (sys_imp_date >= '2024-07-31'
AND sys_imp_date <= '2024-08-06') limit 1000
```

- **作用**: `DetailQueryOptimizer` 显示了在执行 `handleNoMetric` 优化后的 SQL 查询。优化前后的 SQL 并没有变化，这表明在此优化步骤中，SQL 语句已经达到最佳状态，或者该优化步骤对该查询没有进一步的优化空间。

- **SQL 语句**:
  - 与优化前的 SQL 语句相同。