Fink SQL 实践之OVER窗口

问题场景

Flink SQL 是一种使用 SQL 语义设计的开发语言，用它解决具体业务需求是一种全新体验，类似于从过程式编程到函数式编程的转变一样，需要一个不断学习和实践的过程。在看完了 Flink 官方文档中 SQL 部分 ，以及官方提供的 SQL Training 后，觉得自己装备了必杀技准备横扫需求了，这时先来一个简单的营销需求：实时计算今天用户加页面维度的浏览次数，即实时输出PV，下游根据某些规则比如今天用户浏览A页面达到N次触发某种动作。当我使用 group by user 并输出到 kafka 中时，发生异常：`AppendStreamTableSink requires that Table has only insert changes` ，这是因为不带时间窗口的分组聚合不支持流式输出，依赖 Flink Dynamic Tables 的实现原理。修改后加上时间窗口但是不支持实时触发，只在窗口结束时输出一次结果，总之仔细翻了个遍，发现上面资料中都没有类似场景，不会这么简单都实现不了吧?

首先使用标准 SQL 中的 group by 对数据进行分组时，每个分组只能输出一行，其中可以有聚合结果和 group by 的列，如果没有开启 `ONLY_FULL_GROUP_BY` 模式（MySQL 5.7 开始默认开启），还可以输出任意一行值的非 group by 的列，而 Flink 则是始终开启，不支持非聚合列输出。反过来如果我们想要在聚合计算后输出所有明细，即每行明细带上聚合结果，这时只能使用 OVER窗口和它的分组 Partition（还有一个思路是聚合结果和明细再关联查询）。

此外在 Flink 中提供了一系列时间窗口函数，如滚动窗口 Tumble、滑动窗口 Hop、会话窗口 Session 等，可以将时间窗口作为 group by 分组项之一，但是正如前面所说它们都只能在窗口结束时触发计算输出，无法满足实时触发需求，这点也只能通过 OVER 窗口实现。

OVER 窗口介绍

OVER 窗口是传统数据库的窗口函数，它定义了一行记录向前多少行或向后多少行作为一个窗口，以此范围进行分组 Partion 和聚合计算。初次接触 OVER 窗口觉得不好理解，但实际上它是 SQL 标准支持的特性，包括 Oracle、MySQL、PostgreSQL 等主流数据库都已支持，下面是来自 PostgreSQL 文档 3.5. Window Functions 中的介绍：

> A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

其中有具体 SQL 例子辅助理解，此外在维基百科有个更简短介绍 SQL window function ，有了以上基础后再来看 Flink SQL 中的 OVER 窗口，官方文档对 OVER 窗口只有几行描述，SQL Training 中甚至没有涉及，这里我们可以参考阿里云实时计算文档 OVER窗口 ，为了解释后面例子，这里我们只关注其中 Bounded RANGE OVER 类型：具有相同时间值（时间戳毫秒数）的元素视为同一计算行，它对应一个向前有限时间范围的行到该行的一个窗口（因为每一个新到的计算行是该窗口最后一个，以后的还没发生当然无法计算），每一个窗口都会触发一次计算和输出。

OVER 窗口应用示例

首先通过 DDL 定义源数据表和结果表，如下输入是用户行为消息，输出到计算结果消息。

```
CREATE TABLE `user_action` (
    `user_id` VARCHAR,
    `page_id` VARCHAR,
    `action_type` VARCHAR,
    `event_time` TIMESTAMP,
    WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
    'connector.type' = 'kafka',
    'connector.topic' = 'user_action',
    'connector.version' = '0.11',
    'connector.properties.0.key' = 'bootstrap.servers',
    'connector.properties.0.value' = 'xxx:9092',
    'connector.startup-mode' = 'latest-offset',
    'update-mode' = 'append',
    '...' = '...'
);

CREATE TABLE `agg_result` (
    `user_id` VARCHAR,
    `page_id` VARCHAR,
    `result_type` VARCHAR,
    `result_value` BIGINT
) WITH (
    'connector.type' = 'kafka',
    'connector.topic' = 'agg_result',
    '...' = '...'
);
```

场景一，实时触发的最近2小时用户+页面维度的点击量，注意窗口是向前2小时，类似于实时触发的滑动窗口。

```
insert into
    agg_result
select
    user_id,
    page_id,
    'click-type1' as result_type
    count(1) OVER (
        PARTITION BY user_id, page_id
        ORDER BY event_time
        RANGE BETWEEN INTERVAL '2' HOUR PRECEDING AND CURRENT ROW
    ) as result_value
from
    user_action
where
    action_type = 'click'
```

场景二，实时触发的当天用户+页面维度的浏览量，这就是开篇问题解法，其中多了一个日期维度分组条件，这样就做到输出结果从滑动时间转为固定时间（根据时间区间分组），因为 WATERMARK 机制，今天并不会有昨天数据到来（如果有都被自动抛弃），因此只会输出今天的分组结果。

```
insert into
    agg_result
select
    user_id,
    page_id,
    'view-type1' as result_type
    count(1) OVER (
        PARTITION BY user_id, page_id, DATE_FORMAT(event_time, 'yyyyMMdd')
        ORDER BY event_time
```

```
        RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND CURRENT ROW
    ) as result_value
from
    user_action
where
    action_type = 'view'
```

场景三，实时触发的当天用户+页面点击率 CTR（Click-Through-Rate），这相比前面增加了多个 OVER 聚合计算，可以将窗口定义写在最后。注意示例中缺少了类型转换，因为除法结果是 decimal，也缺少精度处理函数 ROUND。

```
insert into
    agg_result
select
    user_id,
    page_id,
    'ctr-type1' as result_type,
    sum(
        case
            when action_type = 'click' then 1 else 0
        end
    ) OVER w
    /
    if(
        sum(
            case
                when action_type = 'view' then 1 else 0
            end
        ) OVER w = 0,
        1,
        sum(
            case
                when action_type = 'view' then 1 else 0
            end
        ) OVER w
    )
    as result_value
from
    user_action
where
    1 = 1
WINDOW w AS (
        PARTITION BY user_id, page_id, DATE_FORMAT(event_time,'yyyyMMdd')
        ORDER BY event_time
        RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND CURRENT ROW
    )
```

此外，OVER 窗口聚合还可以支持查询子句、关联查询、UNION ALL 等组合，并可以实现对关联出来的列进行聚合等复杂情况。

OVER 窗口问题和优化

在底层实现中，所有细分 OVER 窗口的数据都是共享的，只存一份，这点不像滑动窗口会保存多份窗口数据。但是 OVER 窗口会把所有数据明细存在状态后端中（内存、RocksDB 或 HDFS），每一次窗口计算后会清除过期数据。因此如果向前窗口时间较大，或数据明细过多，可能会占用大量内存，即使通过 RocksDB 存在磁盘上，也有因为磁盘访问慢导致性能下降进而产生反压问题。在实现源码 `RowTimeRangeBoundedPrecedingFunction` 可以看到虽然每次窗口计算时新增聚合值和减少过期聚合值是增量式的，不用遍历全部窗口明细，但是为了计算过期数据，即超过 PRECEDING 的数据，仍然需要把存储的那些时间戳全部拿出来遍历，判断是否过期，以及是

否要减少聚合值。我们尝试了通过数据有序性减少查询操作，但是效果并不明显，目前主要是配置调优和加大任务分片数进行优化。