10 | 时间维度聚合计算：如何在长时间窗口上实时计算聚合值?

今天，我们来讨论实时流计算中第二类非常常见的算法，即时间维度聚合值的计算。

在09 课中，我们在讨论流数据操作中的聚合 Reduce 操作时，就用到过时间窗口的概念。当时我们的思路是将流数据划分成一个个的滑动窗口，然后在每个窗口内进行聚合计算。这种做法实际上与传统关系型数据库，在实现聚合计算时使用的算法相同。

但是今天，我们要讨论的“时间维度聚合值计算”，则有了两个非常严格的限制：

1. 需要实时计算返回；
2. 时间窗口很长且数据量很大。

由于这两个限制的存在，现在我们则不得不采用另外一种与 09 课时 中的 Reduce 操作截然不同的思路和方法。而究其原因，一方面，当业务需要实时返回，尤其是要求每条数据在毫秒内返回时，就不再适合使用类似于每次滑动 1 秒的滑动窗口了；另一方面，当窗口非常长，并且数据量很大时，采用窗口计算的方式既需要保存大量数据，还需要对窗口内的数据进行全量计算，这样就不能够实现实时的效果了。

那我们究竟该如何在“长时间窗口且数据量很大”的情况下，实现“时间维度聚合值”的“实时计算”呢？这就是我们接下来要详细讨论的问题。

实时计算时间维度聚合值的难点是什么?

按时间维度对数据进行聚合，是非常常见的计算问题。比如你是一个公司的老板，你想知道公司这个月的运营情况，你肯定是问这个月的销售额和成本各是多少，而不会去问每一笔买卖。

实际开发工作也如此，大部分数据系统的主要工作就是对数据做各种维度的聚合运算，比如计数（count）、求和（sum）、均值（avg）、方差（variance）、最小（min）、最大（max）等。而“流数据”作为一种数据系统，也是如此。

以风控场景为例，我们经常需要计算一些时间维度聚合特征。比如“过去一周在相同设备上交易次数”“过去一天同一用户的交易总金额”“过去一周同一用户在同一 IP C 段的申请贷款次数”等。如果用 SQL 描述上面的统计量，分别如下：

```
# 过去一周在相同设备上交易次数
SELECT COUNT(*) FROM stream
WHERE event_type = "transaction"
AND timestamp >= 1530547200000 and timestamp < 1531152000000
GROUP BY device_id;
# 过去一天同一用户的总交易金额
SELECT SUM(amount) FROM stream
WHERE event_type = "transaction"
AND timestamp >= 1531065600000 and timestamp < 1531152000000
GROUP BY user_id;
# 过去一周同一用户在同一IP C段申请贷款次数
SELECT COUNT(*) FROM stream
WHERE event_type = "loan_application"
AND timestamp >= 1530547200000 and timestamp < 1531152000000
GROUP BY ip_seg24;
```

上面的这些 SQL 让我们很容易想到关系型数据库。关系型数据库在执行上面这类 SQL 时，如果没有构建索引，执行引擎就会遍历整个表，过滤出符合条件的记录，然后按 GROUP BY 指定的字段对数据分组并进行聚合运算。

而当我们面对的是“流数据”时，应该怎样实现这类聚合计算呢？一种简单的策略，是复用上面用关系型数据库实现聚合运算时的方法。

当数据到来时，先把它保存到缓冲区，然后遍历窗口内的所有数据，过滤出符合指定条件的事件，并进行计数或求和等聚合运算，最后输出聚合结果。

但是大多数情况下，将这种简单的方式运用在实时流计算中，十有八九会遇到**性能问题**。

这是因为，**如果将每条消息都保存在缓冲区中，当窗口较长、数据量较大时，会需要占用很多内存。而且每次的计算需要遍历所有的数据，这无疑会消耗过多的计算资源，同时还增加了计算所耗的时间。**

因此，我们需要尽可能地**降低计算复杂度，并且只保留必要的聚合信息，而不需要保存所有原始数据**。

非常幸运的是，对于各种聚合类型的运算，我们都能够找到一个（或者一组）指标，用于记录聚合后的结果。比如，对于 count 计算这个指标是“记录数”，对于 sum 计算这个指标是“总和”，对于 avg 计算这组指标是“总和”和“记录数”，对于 min 计算这个指标是“最小值”，对于 max 计算这个指标是“最大值”。

如果我们用**寄存器**来记录这些指标，那么我们会发现计算每种任务都只需要使用少数几个**寄存器**即可，这就给我们提供了极大的优化空间。

下面，我们以 count 计算来讲解下优化后算法的工作原理。下图 1 是优化后算法的原理图。

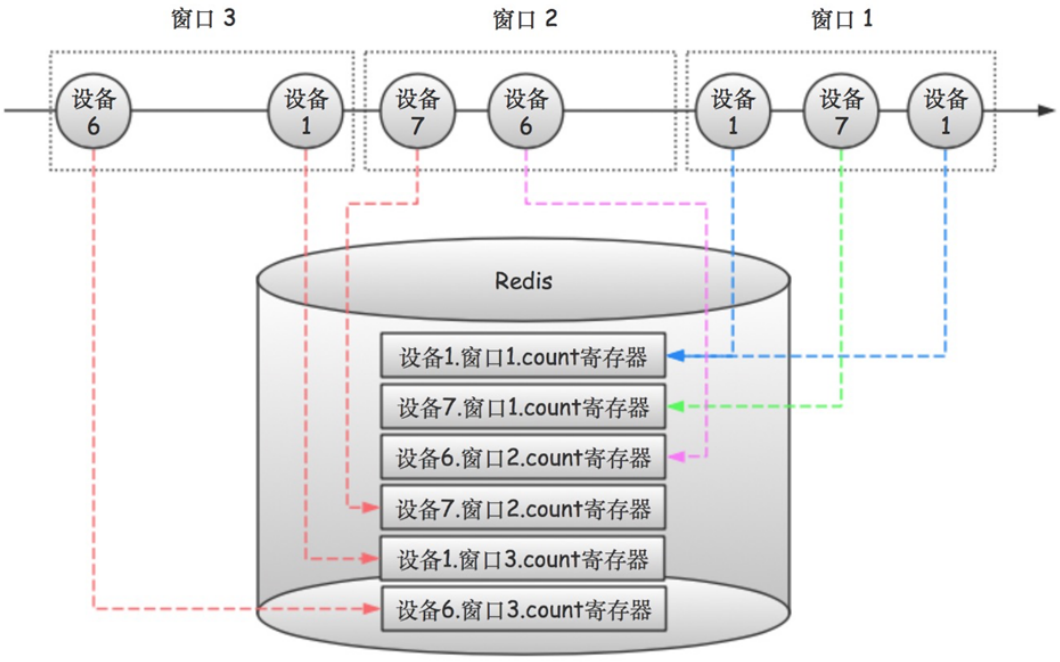

图 1 使用寄存器实现 count 计算原理图

在上面的图 1 中，我们以计算“过去一周在相同设备上交易次数”为例。由于是要计算“过去一周”的时间范围，所以我们将每个窗口设置为 1 天。换言之，图 1 中的窗口 1、窗口 2 和窗口 3 ，都各自代表了 1 天的时间长度。在窗口 1 中，首先出现的是设备 1 上的交易事件，所以我们分配一个名字（对应 Redis 里的 key）为“设备1.窗口1.count寄存器”的寄存器，来记录设备 1 在窗口 1 内交易事件发生的次数。这个 count 寄存器的初始值是 0，每当窗口 1 内来了一个设备 1 上的交易事件时，我们就将这个 count 寄存器的值加 1。这样，当窗口 1 结束时，“设备1.窗口1.count寄存器”的值，就变为了 2。同样，对于其他设备和其他窗口的交易事件，也用相同的方式，分配对应设备和窗口的寄存器，并在每次交易事件到来时，将寄存器的值加 1 。

通过上面的方法，最终我们就可以得到各个设备在各个窗口内的交易次数了。而由于我们的计算目标是“过去一周在相同设备上交易次数”，且每个窗口代表 1 天，所以只需要将连续 7 个窗口内寄存器值读取出来后，累加起来即可得到最终结果了。

以上就是使用寄存器实现 count 计算的整体思路。同样，对于 sum、avg、variance、min、max 等其他类型的时间维度聚合值，都可以按照这种思路来进行计算，只需要先设计好需要使用的寄存器即可。

下面的表 1 就总结了在采用寄存器方法计算各种聚合值时，所需要的寄存器以及各个寄存器的含义。

表 1 各种聚合计算使用的寄存器含义

聚合计算	寄存器1	寄存器2	寄存器3
计数 count	记录数	无	无
求和 sum	总和	无	无
均值 avg	总和	记录数	无
方差 variance	总和	平方和	记录数
最小 min	最小值	无	无
最大 max	最大值	无	无

以上列举的都是我们在平时开发过程中，经常会用到的聚合值。对于其他类型的聚合值，比如偏度（skewness）、峰度（kurtosis）等，通过数学公式转化，也都可以找到对应需要记录的指标，这里就不再展开了。

如何实现时间维度聚合计算

上面说明了时间维度聚合值计算的整体思路。那具体应该怎样实现呢？这里我使用 Redis 并结合伪代码的方式来详细讲解下。

与前面讲解 count 计算原理时一样，我们要计算的时间维度聚合值还是“过去一周在相同设备上交易次数”。

针对这种计数查询，非常适合用 Redis 的 INCR 指令。INCR 是 Redis 中经常会被使用到的指令，它可以对存储在指定键的数值进行“原子加一”，并返回加一后的结果。

这里我们将 7 天的时间窗口划分为 7 个小窗口，每个小窗口代表 1 天。在每个小窗口内，分配一个 key 用来记录这个窗口的事件数。 key 的格式如下：

```
$event_type.$device_id.$window_unit.$window_index
```

其中，“$event_type”表示事件类型，“$device_id”表示设备 id，“$window_unit”表示时间窗口单元，“$window_index”表示时间窗口索引。

比如，对于“device_id”为“d000001”的设备，如果在时间戳为“1532496076032”的时刻更新窗口，则计算如下：

```
$event_type = transaction
$device_id = d000001
$window_unit = 86400000  # 时间窗口单元为1天，即86400000毫秒
$window_index = 1532496076032 / $window_unit = 17737    # 用时间戳除以时间窗口单元，得到时间窗口索引
$key = $event_type.$device_id.$window_unit.$window_index
redis.incr($key)
```

上面的伪代码描述了使用 Redis 的 INCR 指令更新某个窗口的计数。我们将更新操作和查询操作分开进行，因此这里只需更新一个小窗口的计数值，而不需要更新整个窗口上所有小窗口的计数值。

当查询 7 天窗口内的总计数值时，我们对 7 个子时间窗口内的计数做查询并汇总。计算如下：

```
$event_type = transaction
$device_id = d000001
$window_unit = 86400000  # 时间窗口单元为1天，即86400000毫秒
$window_index = 1532496076032 / $window_unit = 17737    # 用时间戳除以时间窗口单元，得到当前时间窗口索引
sum = 0
for $i in range(0, 7):
    $window_index = $window_index - $i
    $key = $event_type.$device_id.$window_unit.$window_index
    sum += redis.get($key)
return sum
```

在上面的伪代码中，用 Redis 的 GET 指令，查询了过去 7 个子时间窗口，也就是过去 7 天每天的计数，然后将这些计数值汇总，就得到了我们想要的“过去一周在相同设备上交易次数”这个特征值。

寄存器方案的不足之处

虽然说，采用寄存器的方案，极大减少了内存的使用量，也降低了计算的复杂度，但是这种方案依旧存在问题。由于采用了“寄存器”来记录聚合计算的中间值，也就涉及“状态”的存储问题。

或许乍看之下我们会觉得，寄存器嘛，无非存储一个数字而已，又能够占用多少空间呢？但稍微仔细分析下就会发现问题了。

我们为变量的每个可能的值都分配了一个或一组寄存器，虽然寄存器的个数不多，比如在表 1 中使用寄存器最多的方差也就用了 3 个寄存器。当我们进行聚合分析的变量具有一个较低的“势”时（“势”是集合论中用来描述一个集合所含元素数量的概念。比如集合 S={A, B, C}有 3 个元素，那么它的势就是 3。集合包含的元素数量越多，其势越大），那么一切都尚且安好。

但是，实际的情况是，我们**用于分组聚合时的分组变量，往往具有比原本预想高得多的势**。比如统计“用户每天的登入次数”，那全中国有十四亿人口！再比如需要统计“每个 IP 访问网站的次数”，那全球有四十多亿 IP ！再加上，有时候我们需要聚合的是一些复合变量，比如统计“过去一周同一用户在同一IP C段申请贷款次数”，这种情况如果严格按照理论值计算，需要采用笛卡尔积，那将是天文数字了。

所以，至少我们不能指望将这些状态都存放在本地内存里。通常，我们**需要将这些寄存器状态保存到外部存储器，比如 Redis 、Apache Ignite 或本地磁盘中**。并且，我们还需要为这些状态设置过期时间（TTL），将过期的状态清理掉，一方面为新的状态腾出空间，另一方面也避免了占据空间的无限增长。

“状态”存储其实是一个非常重要的问题，而且在后面讨论其他几类算法时，也都会涉及有关“状态”存储的问题。所以，这里我只是先将“状态”存储问题和初步解决思路给了出来，在后面的课时中我们还会针对流计算中的“状态”问题做专门讨论。

小结

今天，我们讨论了实时流计算中第二类算法问题，即时间维度聚合值的计算。

应该说，正是因为“实时计算”和“长周期窗口”这两个前提条件，共同决定了我们必须采取“寄存器”的方式，来优化时间维度聚合值的算法。而“寄存器”的引入，则使得流计算变成了“有状态”的系统，这也直接导致了各种开源流计算框架专门引入“状态存储”相关的功能，并提供了对应的 API 编程接口。

所以，以后你在使用诸如 Flink、Spark Streaming 这样的流计算框架，遇到状态相关的 API 时，一定要清楚它们为何而来，并将它们灵活地用于你的业务实现中。

最后，我们今天是用 Redis 实现的“过去一周在相同设备上交易次数”，那如果是使用 Flink 来实现这个时间维度聚合值计算，你会怎么做呢？注意要求是针对每一个交易事件进行计算，并将计算结果附加到交易事件上组成新的事件，最后再将这个新事件作为流数据输出。另外，计算过程中，你可以使用 Fastjson 库的 JSONObject 对象表示事件。思考并试验下，可以将你的思路或问题写在留言区！

下面是本课时内容的脑图，可以帮助你理解。