文章编号：2096-1472(2023)-04-42-04
DOI:10.19644/j.cnki.issn2096-1472.2023.004.010

流式计算引擎中密集滑动窗口的性能优化研究

程盛阳

(中国银联股份有限公司，上海 201201)
✉chengshengyang@unionpay.com

摘　要：为缓解目前的大数据流式计算引擎在处理密集窗口时因高负载而带来的性能下降问题，文章分析了原生窗口机制的性能瓶颈以及现有优化方法的不足之处，包括需要额外的内存空间用于存储输入的数据流、无法自动清理状态缓存等，提出一种基于关键窗口机制的优化方案，该方案能够减少流式计算中需要创建的窗口数量，具有降低系统负载的效果。通过与原生机制进行对比分析，证明此优化方案的有效性。该优化方案具有能兼容现有框架、对下游系统改造少及同时提升内存占用和I/O频率两个方面性能的优点。

关键词：大数据；流式计算；窗口计算；Flink
中图分类号：TP316.4　**文献标识码**：A

Research on Performance Optimization of Dense Sliding Windows in Streaming Computing Engines

CHENG Shengyang

(*China Unionpay*, *Shanghai* 201201, *China*)
✉chengshengyang@unionpay.com

Abstract: In order to alleviate the performance drop caused by high load of current big data streaming computing engines when processing dense windows, this paper proposes to analyze the performance bottleneck of the native window mechanism. And the shortcomings of some existing optimization schemes are pointed out as well, including the need for additional memory space to store the input data stream, and the inability to automatically clean the state cache. Then, an optimization scheme based on key-window mechanism is proposed, which can reduce the number of windows to be created in streaming computation and therefore reduces the system load. The effectiveness of this optimization is shown by a comparative analysis with the native mechanism. This optimization scheme has the advantages of being compatible with existing frameworks, requiring little modification of downstream systems, and enhancing both memory and I/O performance.

Keywords: big data; streaming computing; window computing; Flink

1 引言(Introduction)

在大数据技术发展早期，批处理技术应用广泛，如阿帕奇软件基金会开发的Hadoop MapReduce框架、加州大学伯克利分校开发的Spark框架等，取得了令人瞩目的成果[1]；但随着业务要求的不断提高，离线计算高延迟的弊端逐渐暴露，流式计算引擎应运而生，包括最早由推特公司开发的Storm框架、Spark框架的流计算扩展Spark Streaming、谷歌公司开发的Google Dataflow框架、最早由柏林工业大学开发的Flink框架等[2]，它们能在数据连续到达的同时进行实时计算，被广泛应用在对时间性要求很高的场景中。流式计算的数据源没有边界，由计算引擎负责确定窗口范围，但在如“双11”网购促销日、春运抢票等高负载应用中，原生窗口的性能常无法满足实际应用需要[3]。分析原因，一是这些框架采用的全量创建窗口的方式难以支持毫秒级的刷新频率，生成的窗口数量巨大；二是交易数据流存在非均匀性，为及时计算活跃用户的数据，窗口必须密集，但也会导致系统为低活跃的用户构造大量的重复窗口，造成资源浪费。

本文以由阿帕奇软件基金会孵化的Apache Flink计算引

擎为例，分析其窗口机制性能缺陷的根源。之后提出一种优化密集滑动窗口的方案，能减少系统需要构造的窗口数量，并通过计算比较两种方案，阐明优化方案的有效性。

2 原生窗口机制及其问题(The native window mechanism and its problems)

2.1 原生窗口机制

流式计算中的数据源不断产生数据形成流，需要由窗口划定一段时间范围的计算结果。例如，一个商务平台中对每个商户的交易进行1 min的统计，每500 ms更新一次计算结果，生成一个滑动窗口，则1 min为滑动窗口的长度(*size*)，500 ms为此窗口的步长(*slide*)，商户号为此窗口的聚合键(key)。

在理想情况下，流处理引擎应该为每个key预先分配好滑动窗口，这样在有数据到达时，数据就可以直接落到对应的窗口中，但实际上流式计算框架普遍采用懒构造方式[4]，这种方式为了节省资源是不会为还未出现的key预先分配窗口的，只有当一个key有对应数据到达后才会创建，然后向前追溯，补充生成之前由于懒策略而跳过的所有窗口。图1展示了Flink计算滑动窗口的创建过程：数据在*t*时刻到达后，除了创建从*t*到*t*+*size*范围的窗口，还通过循环不断向前补充创建需要追溯的滑动窗口。

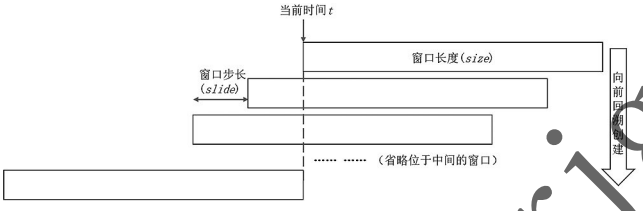

图1 原生窗口机制示意图

Fig.1 Diagram of the native window mechanism

在后续不断有新数据抵达时，窗口构建策略依然不变，继续循环和向前追溯。此外，Flink会将准备创建的新窗口和已有的窗口进行比较，合并相同的窗口，这在Flink源代码中streaming.runtime模块的WindowOperator类的processElement方法中实现。

2.2 滑窗机制存在的问题

首先是数据倾斜带来的问题，在流式系统中的数据源是非均匀性的，在相同时间内，不同key产生的数据量级存在显著的差异，或者对于同一个key，其数据产生频率在不同时间段存在显著的差异。对于频繁更新的key，为了保证数据的及时性，必须使用较小的步长，满足热点key的刷新频率要求，导致系统不得不为稀疏key也配置同样的窗口创建策略，而这些窗口内部大部分都保存了相同的状态、具有相同的输出，实际上是多余的，造成内存资源的浪费。

其次是实时性问题，在很多系统中，响应时间是评估系统能力的硬指标，例如对于风险监控系统，风险行为越早被检测到，被拦截或挽回损失的可能性就越大。流式计算引擎的窗口刷新频率决定了一个风险行为从发生到体现在计算结果的延迟时间。假设窗口计算的步长是10 min，那么无论将系统处理和数据传输的延迟压缩到多低，在最坏情况下，一个风险事件也要在10 min后才从窗口统计中被输出。在这个时间差之内，系统无法感知到风险的发生，也无法及时响应。因此，理想情况下，毫秒级的窗口步长是最佳的，但根据窗口的定义方式，对每个新key所需要创建窗口的个数=窗口长度/窗口步长，当窗口长度达到小时甚至天数时，维持毫秒级的步长会导致窗口数量巨大，如果实际系统中有百万级以上数量的key，就会带来大型分布式计算架构也难以承担的负载。

同时，密集的多余窗口在创建和销毁时的高并发会导致CPU占用过高和I/O负载高，这使得高频(毫秒级延迟)的窗口很难用原生窗口实现，因为在下一轮窗口创建时，上一轮窗口产生的CPU占用和I/O负载可能还未被完全消化，导致系统性能雪崩问题[5]。

3 现有优化及其不足(Existing optimization and its deficiencies)

3.1 ProcessFunction优化方法

应用层目前使用处理函数(ProcessFunction)替代传统窗口是一种常用的优化方法，它本质上是应用一个流处理函数，在其中定义对每条上游数据的独立处理逻辑[6]。ProcessFunction对一个key只构造一个实例，接受与保存新到达的数据，不断地对过期的旧数据进行清理。以Flink为例，其提供了MapState类用于保存ProcessFunction的状态，之后可以调用windowState.put()方法向其中添加和更新状态。这样就可以将ProcessFunction当作一个窗口，虽然此优化避免了窗口冗余，但是要求在函数内部保存原始数据，因此削弱了优化效果。此外，如果Flink上游数据更新非常快，而ProcessFunction没有滑窗策略的频率控制的步长，下游的I/O负载压力会显著增大。

3.2 采用分桶策略的ProcessFunction优化

为了减少ProcessFunction优化方法中保存原始数据过多消耗的内存空间，可以在牺牲一定精度的情况下采用分桶策略，在新数据到达时进行部分聚合。例如，对一个长度为10 min的窗口，每分钟分为一桶，这样窗口中只需要存10 份子状态，每分钟清理一个桶即可，不再需要保存原始数据。图2展示了采用分桶策略的ProcessFunction优化方法的原理。

图2 分桶策略的ProcessFunction优化示意图

Fig.2 Diagram of ProcessFunction optimization with bucket splitting strategies

此优化方法虽然在一定程度上解决了内存问题，但同时限制了计算的粒度，如果分桶的间隔过小(例如毫秒级分桶)，就难以达到优化效果，甚至因加入了额外的逻辑而进一步加重了系统的负担。扩大分桶的间隔相当于放大了滑动窗口的步长，虽然能够起到优化内存占用的效果，但是无法满足上文所述的对于高刷新频率的需求。

综上所述，应用层的常用优化方法虽然能够在一定程度上减少窗口冗余、降低一定的内存负载，但其优化策略均存在一定的副作用，难以满足非均匀数据源、高窗口刷新率场景的应用要求。除此以外，使用这些优化方法不仅要求重新编码实现计算逻辑，还必须手动维护和清理状态，这与直接使用引擎提供的窗口功能相比，额外增加了开发工作量。

4 基于关键窗口机制的优化(Optimization based on key-window mechanism)

为解决非均匀数据源中稀疏key导致的状态内存资源浪费及CPU占用高和I/O负载高的问题，本文提出一种基于关键窗口机制的窗口实现优化方案，即在不影响计算结果正确性的情况下，流处理引擎只进行真正组成计算结果的关键窗口的创建，省略多余的原生窗口，从而较少了内部窗口数量，优化了系统性能。

4.1 关键窗口的定义

如上文所述，流式处理引擎的原生窗口机制的问题根源在于，为稀疏key构造了大量的内部状态和输出都相同的冗余窗口。为了从根本上优化系统性能，就需要设计一种方法使得系统跳过对多余窗口的创建。在排除冗余窗口之后，剩下的窗口就是关键窗口。

具体可以将“关键窗口”定义为所有窗口中那些输出能使得计算结果实际产生变化的窗口。例如，对于图3中的数据序列，每个方格表示一个时间单位，方格中的圆形符号表示该时间单位内上游系统向流计算引擎发送了数据。

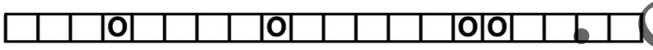

图3 流式数据序列

Fig.3 Streaming data sequence

在本研究中，采用长度为4 个单位时间、步长为1 个单位时间的滑动窗口进行统计，如果采用原生的窗口逻辑，则需要创建的窗口如图4所示，图中数据流上下方的每个线段表示一个窗口，共有13 个。

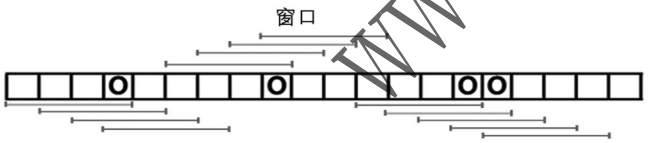

图4 原生策略下需要创建的窗口

Fig.4 Windows to be created in the native mechanism

对同样的数据源若采用优化的策略，根据上文对“关键窗口”的定义，需要创建的关键窗口如图5所示，共有8个。这些关键窗口分别对应了各个计算结果发生变化的时间节点：“1”表示首个数据点进入统计，“2”表示首个数据点离开统计，“3”表示第二个数据点进入统计，“4”表示第二个数据点离开统计，“5”表示第三个数据点进入统计，“6”表示第四个数据点进入统计，“7”表示第三个数据点离开统计，“8”表示第四个数据点离开统计。

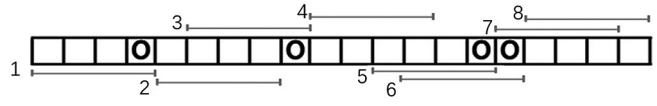

图5 优化策略中定义的关键窗口

Fig.5 Key windows defined in the optimized mechanism

在本研究中，关键窗口的数量比默认窗口少5 个，而随着窗口长度越长、窗口步长越短，以及数据源的不均匀程度越高，关键窗口和原生窗口相比，在数量上的优势就会越发明显。

4.2 关键窗口优化方案的实现

本文提出的关键窗口优化方案，不需要推倒现有的流计算框架从零设计新系统，而是可以通过对现有的流计算引擎进行适当的改造，使其支持关键窗口的创建和计算。这样能充分利用成熟引擎在功能和稳定性上的诸多优势，使优化方案低成本地投入工程应用。

改造现有流计算引擎实现关键窗口，关键在于使关键窗口的创建和计算都基于原生的方法实现，不增加额外的方法和状态存储。以Flink为例，其原生的窗口机制是基于Accumulator(也称Aggregate Function，聚合函数)的方式实现的[7]，这要求在仅利用窗口的add()、merge()和getResult()方法的前提下，完成对所有关键窗口的生成和计算，而不需要借助额外的Evictor(即删除元素的方法)或其他的状态管理手段：不使用Evictor，是因为指定Evictor之后，会使得窗口不再进行预聚合，实际上导致系统丧失了有状态计算的优势；而借助其他底层状态管理函数，如ProcessFunction，则会丢失窗口的语义性，使其和更高抽象的API(如SQL API)的兼容出现问题[8]。

基于上述前提，本文提出的关键窗口实现方案如下。

将关键窗口分为左关键窗口和右关键窗口，关键窗口定义示意图如图6所示。左关键窗口是有数据进入而使得计算结果发生改变的窗口，右关键窗口是有数据离开而使得计算结果发生改变的窗口。

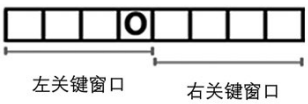

图6 关键窗口定义示意图

Fig.6 Diagram of key window definition

对窗口创建逻辑的改变不影响数据进入窗口后的计算方式，需要相应改变的是窗口状态的初始化方法：在传统的窗口机制中，窗口的创建是时间驱动的，窗口按照步长逐个创建，新数据到达时，它所属于的所有窗口在逻辑上已经处于预备的状态，在等待这个新数据；而在关键窗口机制中，窗口的创建是数据驱动的，所有窗口均是随着相对应的某个数据点的到达而创建的。因此，在新数据到达时，它一方面会落入此前已经创建好的窗口，另一方面同时需要创建其自身所对应的关键窗口。

右关键窗口反映的是这个数据点失效而使计算结果发生的变化。因为在创建时，当前数据点是最新的，其后的数据还未到来，所以为了表示当前数据点失效而产生的变化，直接创建一个空窗口即可。

左关键窗口反映的是这个数据点加入统计后使得计算结果发生的变化。左关键窗口在创建时需要继承此前的历史状态。根据对“关键窗口”的定义，系统会在有数据点进入和

失效的关键时间位置创建窗口，而系统的统计结果是完全由窗口输出的。因此，当一个新数据点抵达时，必然能够找到一个已经存在的窗口，其中保存的状态正是新窗口需要继承的历史状态。最简单的一种情况是，在前一个窗口时间内，没有其他数据抵达，即无历史状态可以继承，这种情况下直接创建空窗口即可。如果在前一个窗口时间内存在数据点，就要将当前的数据合并到最新的历史状态中，即取到系统输出的前一个统计结果，再将新数据加入计算。在流计算引擎中，窗口管理器只记录各个窗口起始和结束的位置信息[9]，它无法直接感知到系统的前一个统计结果是什么，或者是由哪个窗口触发的，因此需要通过判断窗口位置的确定从哪个窗口继承历史状态信息。

具体策略如下：当新数据到达时，如上文所述，首先创建一个空的右关键窗口；然后确定此数据对应的左关键窗口的位置；检查即将创建的左关键窗口和此前存在的左关键窗口是否存在重合；若无重合，直接创建此窗口；若有重合，检查即将创建的左关键窗口与最近一个左关键窗口的非重叠部分是否包含数据(即其中是否存在右关键窗口的起点)；若没有这样的右关键窗口，调用merge()方法，合并最接近的一个左关键窗口；若存在这样的右关键窗口，调用merge()方法，合并最新的一个右关键窗口。

对下游使用计算结果的系统而言，采用关键窗口后，下游系统依然能够感知到所有计算结果的更新，而唯一的区别在于，原生窗口会每隔一个窗口步长的时间就向下游发送一次数据，而关键窗口仅在计算结果更新时发送。为了配合优化，下游系统需要对失效时间的利用做简单的调整。

具体来说，在使用原生窗口时，引擎不会发送某个key统计结果归零的数据，需要下游系统自行观察该key的数据不再更新，从而得知该key的最后一个统计结果过期。在使用关键窗口优化后，数据过期时会将全窗口的输出发送给下游，因此下游系统仅需将数据过期时间调整为窗口长度即可，如果下游系统在此数据过期之前收到了新的输出，则说明有新数据抵达，要对数据过期时间进行重置。因此，改用关键窗口后，仅需调整下游系统的数据过期时间即可兼容此优化，不会影响系统本身的功能。

4.3 优化效果

设某系统正在使用长度为p s，步长为k s的窗口。对目前的流式计算引擎而言，接收到n 个数据后，需要创建的窗口最大个数为$\frac{p\times n}{k}$ 个。这是因为每接收到一个数据，系统都需要创建最多$\frac{p}{k}$ 个窗口。在引入关键窗口优化后，接收到n 个数据后，系统需要创建的窗口数量为$2n$ 个。这是因为关键窗口仅体现数据加入统计、失效离开统计的变化，系统每接收到一个数据，只需要创建2 个关键窗口。

在实际应用中，$\frac{p}{k}$几乎总是大于2的。例如，统计每个用户在过去120 s内的操作次数，每2 s更新一次统计结果，那么$\frac{p}{k}$=60。显然，统计窗口的长度越大、刷新频率越快时，$\frac{p}{k}$值也就越大，将远远超过2。可见，关键窗口优化能够有效地减少需要的窗口数量。

5 结论(Conclusion)

首先，本文分析了目前流式计算引擎在密集窗口情况下在性能方面存在的潜在问题，指出采用原生的窗口策略是导致高负载的原因。其次，说明了现有的优化方法虽然能降低内存消耗，但是存在不足之处。最后，本文提出了基于关键窗口的优化方案，通过减少计算中创建的窗口数量，能有效地降低系统在内存和I/O两方面的性能压力。目前，本优化方案是基于系统时间的，若要推广到使用事件时间的应用，未来可以进一步优化对数据流水位线等机制的兼容。

参考文献(References)

[1] DEAN J, GHEMAWAT S. MapReduce: Simplified data processing on large clusters[C]// CACM. Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation. New York, USA: Association for Computing Machinery, 2004:137−150.

[2] 孙大为,张广艳,郑纬民.大数据流式计算:关键技术及系统实例[J].软件学报,2014,25(4):839−862.

[3] 李圣,黄永忠,陈海勇.大数据流式计算系统研究综述[J].信息工程大学学报,2016,17(1):88−92.

[4] 毕倪飞,丁光耀,陈启航,等.数据流计算模型及其在大数据处理中的应用[J].大数据,2020,6(3):73−86.

[5] 代明竹,高嵩峰.基于Hadoop、Spark及Flink大规模数据分析的性能评价[J].中国电子科学研究院学报,2018,13(2):149−155.

[6] WU F, FU M. Stream processing[M]. Heron Streaming: Fundamentals, Applications, Operations, and Insights, 2021: 3−14.

[7] TRAUB J, GRULICH P M, CUELLAR A R, et al. Scotty: general and efficient open−source window aggregation for stream processing systems[J]. ACM Transactions on Database Systems, 2021, 46(1):1−46.

[8] SAYED F R, KHAFAGY M H. SQL to Flink translator[J]. International Journal of Computer Science Issues, 2015, 12: 169−174.

[9] 施国欢,宋吉,李江华.不同特征的流数据对Flink性能影响研究[J].计算机科学与应用,2022,12(11):2599−2607.

作者简介：

程盛阳(1996−)，男，硕士，助理工程师.研究领域：计算机应用.