Sliding Window Operation Improvement

Goals

The goal of this document is to layout a roadmap planning for the improvement of Flink's window operator - in order to support (1) current use cases more efficiently; and (2) extended new use cases, of windowing operation based on various previous discussions (mailing list, JIRA, etc).

This document is the first phase implementation planning for the overall proposal **[D1]**. It is targeted to improve most of the pain points discussed in **[T1]**.

Scope of Use Cases

In-scope Use Cases

In-scope current use cases

The following are use cases we would like to target which experienced poor performance currently. They are mostly discussed in the original sliding window optimization JIRA **[J1]**, and should be supported ***without any*** public DataStream API changes.

[IU1] Full buffering of incoming elements across multiple windows, with rich or process window function
The first type of current use cases requires incoming elements to be fully buffered across multiple windows as is. The reason is that the type of window functions (rich, process) used does not have efficient incremental “aggregation”.

[IU2] Computation of incoming elements across multiple windows, with incremental aggregation.
This type of current use cases requires incoming elements to “incrementally” buffer into the appending state, such as “Reduce” or “Aggregate” functions. Due to the fact that efficient incremental aggregate is possible, it is better to keep the current multi-window buffering approach.

[IU3] Computation of incoming elements across multiple windows, with efficient merge window function
This type of current use cases requires incoming elements to be computed across multiple windows. However the performance is poor, depending on the overlapping nature of the multi-window span.

Out-of-scope Use Cases

In order to deliver a more actionable, executable design doc, the following are not in scope of this discussion. We will evaluate these use cases as a next step.

Out-of-scope new use cases

The following use case are some of the ones discussed in the new mailing list thread **[T1]** and in the discussion on **[J2]**. These use cases ***“may”*** require new public DataStream API changes.

[OU1] Full buffering of incoming elements that fires computation with neighboring elements
This type of new use cases requires no collapse based on a count or time range; instead, it collects output result for incoming elements based on its neighboring elements (Defined similarly with SQL as OVER window aggregate). However, the triggering rule can be more flexible in DataStream perspective.

[OU2] Full buffering of incoming elements that fires computation with neighboring elements, with efficient retracting window function.
Similar to [OU1], in addition, this type of use cases contains operator with efficient retracting methods (see Table aggregation function for example)

Out-of-scope complex existing use cases

[OU3] Definitively non-overlapping windows
These types of use cases were discussed in **[D1]** as one of the main use cases, however in order to support such characteristic, a major change of public API is required with very little gain.

[OU4] Concatenating windows
These types of use cases were discussed in **[D1]** as one of the futuristic use cases. This involves a sequence of windowing operation over the exact same key. This would not be efficient on current API as the windowing results are interpreted as normal DataStream. For these type of use cases to work properly, a modification of API is required.

[OU5] Multi-timeout session windows
These types of use cases were discussed in **[D1]** as one of the futuristic use cases. This involves a parallelly operating window operators with session assigners that contains different timeout periods. For these type of use cases to work properly, a modification of API is required.

Prior Arts

Blink WindowOperator

[A1] Blink window operator with aggregate functions

Blink's window operator used in flink-table package (link) has specific improvement used to optimize windowing operation on aggregation functions. These includes:

1. Specific internal version of the WindowFunction interface that extends to support not only the process element, but also interacts with window assignments. (assignActualWindows, assignStateNamespace, etc)
2. Window states designed for retraction when available. (subkeystates, etc)
3. Much simpler design (no merge window)

Due to the fact that most of these operations are designed for aggregate functions, we would not be able to reuse directly in DataStream API. However, some of the component we described above should be leveraged.

Window Slicing POC

[A2] Window with internal slicing state

Window slicing POC #1 (link) focus on directly alter the windowing state: split the state into slicing state + slice-to-window state.

This approach is very similar to **[A1]** approach with paned window functions (by creating the concept of actual window and affected window). However, this POC is designed for no only aggregate function, but all window functions including reduce, fold as well as a more generic window apply (via Fluent iterables).

This POC approach requires minimum public API changes in order to make the slicing work properly. However, backward-compatibility of internal state is a problem since the internal state is splitted into 2 parts.

[A3] Window with external slicing and merging step

This approach (link) focus on explicitly allowing users to control the "state" as well as the "windowing" operation.

Instead of using WindowedStream compiles user-defined windowing function into “state” appending step and window processing step, it removes such complexity by directly letting user configure the 2 steps separately.

This POC approach requires significant amount of API changes, and required some simplification on the DataStream windowing API. However, it does not introduce any backward-compatibility issues.

Scope

Observations from Prior Arts

The basic observation from previous experimentations is that:

- There’s no one-solution-fits-all optimization mechanisms that can be used to optimize sliding window directly **[A1,A2]**. Performance experiment indicated that the result of the optimization depends highly on
  - (1) the pattern of the window.
  - (2) the pattern of the incoming traffic.
  - (3) the complexity of the window process function.
- There are use cases that cannot be covered by current windowing API: They require extra amount of flexibility for users **[A3]**.
  - Same-window operations
  - Cascade window operations
  - Special operations that requires more granular control of the relationship of window state and window function.

Problem Statements

From the observation we conclude that:

- The current window operator hides the complexity of how a specific window function is optimized in an window operator by: Splitting the operations into (1) efficient internal window state; (2) internal window function that process the internal window state.

The problem statement for this design doc is: How to provide better optimization of the window operator on “sliding window”. As we’ve discussed in the usage scope and observation sections: we will only target **[IU1, IU2, IU3] without public API changes.**

Design

Overview

The overall design consists of 2 major parts: Runtime/Operator Improvements and API improvements.

Runtime/Operator Improvements

The runtime/operator changes will be pretty substantial in order to utilize a more efficient windowing mechanism, we will break down the improvement design into 3 parts.

[DR1] Full Iterable buffer Improvement

The first design is to improve the window function where full iterable buffers are generated per window **[IU1]**: this is especially costly in memory when trying to utilize against multiple overlapping windows.

The way we would like to improve this particular use case is to introduce ***“affected window”*** concept proposed in **[A1]** to only record elements in the “actual window” instead of the entire set of overlapping window collection.

When invoke window processing, all element partitions in “affected windows” are “grouped” together and send through the processing method of the window function.

[DR2] Paned incremental aggregation

The second design is a continuation of the work described in **[DR1]**. With the possibility to assign “affected window”, it is also applicable to other window functions incremental aggregations, provided that an ***efficient merge method*** exists.

In order to support more general “buffering” method, we would like to introduce “***affected state***” concept. Similar to affected window: upon receiving the elements for the affected window, operator should invoke internal window function to process and update the affected state.

With this design, the **[DR1]** approach is merely a special case of **[DR2]** with an *iterable affected state*.

However, the design should consider how to deal with the two types of approaches, corresponding to **[IU2, IU3]** :

1. Directly incremental aggregation towards all assigned windows.
2. Only incremental aggregate against the “affected window”, and the use merge operation to compute the overall results similar to how we invoke processing method in **[DR1]**.

See **[DA1]** for more details on this.

Internal API Improvements

Although we would like to make minimum to no changes with public facing APIs in DataStream, there are some modifications needed in order to support **[RD2]**. We propose to have these changes only affects the @Internal APIs.

[DA1] Extension of Internal Window Function [IN DISCUSSION]

Internal window functions are currently used during WindowOperator generation. It splits the actual user function into (1) one that used in the AppendableState that is used to buffer or incrementally aggregate incoming elements; and (2) one that is used to process the AppendableState as the window function itself.

In order to create pane-based, or slice-based iterable buffer Improvement, and indicator is required to indicate which “slice” is currently being updated upon incoming processElement.

Comparing with the approach **[A2]** used to add assigner public API, this is implemented in **[A1]** elegantly without changing the public API or adding any special window assigner: Flink internal can control the ***operator process***, and the ***internal state*** utilization.

In addition, this also makes the implementation piece easier, especially in the backward compatibility section.

Currently, Flink controls how internal window functions are utilized in WindowedStream.

- For **[IU1]**, the approach is to improve all the fully iterable buffering operators.
- For **[IU2,IU3]**, it is not directly obvious which internal window function should be used under circumstances. More details will be discussed in the implementation section.

Implementation Plan

The implementation plan of this is as following:

1. Improve **[IU1]** scenarios where full buffering of window elements are inevitable by reusing pane iterable buffering in **[DR1]** for WindowOperator
2. Support EvictingWindowOperator for **[DR1]**.
3. Introduce proper internal window function described in **[DR2]** that works with **[IU3]** and at the same time make sure existing internal window functions continue to support **[IU2]**.
4. [IN DISCUSSION] Introduce indicator **[DA1]**, that indicates whether windowing operator should be done as either **[IU2]**, or **[IU3]**.

Context Summary

Discussion Threads

[T1] **Window improvement using slicing discussion**

Design Docs

[D1] **Optimize Window Operator with Slicing**

JIRAs

[J1] **FLINK-7001**
[J2] **FLINK-11276**