KIP-770: Replace "buffered.records.per.partition" & "cache.max.bytes.buffering" with "{statestore.cache}/{input.buffer}.max.bytes"

Status

Current state: Partially Done in 3.4: new cache size config and metrics

Discussion Thread: here

JIRA: **Unable to render Jira issues macro, execution error.**

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

The current config "buffered.records.per.partition" controls how many records in maximum to bookkeep per partition, and hence when this value is exceeded for any subscribed partition, the consumer pauses this partition However this config has two issues:

- It's a per-partition config, so the total memory consumed is dependent on the dynamic number of partitions assigned.
- Record size could vary from case to case.

The above 2 reasons makes it hard to bound the memory usage for this buffering.

Proposed Changes

This KIP proposes to deprecate the per partition (**buffered.records.per.partition**) and introduce a new one called **input.buffer.max.bytes**. This would be a global config applicable per Kafka Streams instance. This parameter controls the number of bytes allocated for buffering. Specifically, for a processor topology instance with `T` threads and `C` bytes allocated for buffering, each thread will have an even `C/T` bytes to construct its own buffer. Following points describes some of the characteristics of the new config:

- Whenever all the tasks belonging to a particular Stream Thread have amassed bytes more than what has been allocated to it, the underneath consumer would pause consuming from all partitions that have some data already. What this also means, is that partitions with empty buffers wouldn't be paused.
- The paused partitions would be resumed after the buffered size for all tasks in the thread are lesser than the bytes allocated.
- Whenever there is any modification in the number of Stream Threads in the topology, the buffer bytes allocation to all Stream Threads would change. This behaviour is exactly similar to **`cache.max.bytes.buffering`.**
- As per the suggestions on the dev mailing list, we would also be renaming the config **cache.max.bytes.buffering** to **statestore.cache.max.bytes** as part of this KIP.
- Adding a new metric at a partition-group class level to measure all bytes aggregated per task(**input-buffer-bytes-total)**. This would represent bytes aggregated by all partitions of the input task.
- Adding a new metric at a partition-group class level to measure cache size accumulated per task(**cache-size-bytes-total)**. This would represent bytes aggregated by all partitions of the input task.

Public Interfaces

- Adding a new config **input.buffer.max.bytes** applicable at a topology level.
    - Importance: **Medium**
    - Default Value: **512 MB.**
- Adding new config **statestore.cache.max.bytes**. Importance and default value would be the same as **cache.max.bytes.buffering.**
- Adding new metric called **input-buffer-bytes-total .**
    - type = stream-task-metrics
    - thread-id = [thread ID]

    - task-id = [task ID]
    - The recording level for all metrics will be **DEBUG**
    - **Description**: The total number of bytes accumulated by this task
- Adding new metric called **cache-size-bytes-total .**
    - type = stream-task-metrics
    - thread-id = [thread ID]
    - task-id = [task ID]
    - The recording level for all metrics will be **DEBUG**
    - **Description**: The cache size in bytes accumulated by this task.

Compatibility, Deprecation, and Migration Plan

- The per partition config **buffered.records.per.partition** would be deprecated.
- The config **cache.max.bytes.buffering** would be deprecated.

Rejected Alternatives

- Having the config **input.buffer.max.bytes** at a global level and not distributed across Stream Threads. In this case, all the bytes aggregated across all tasks is calculated at an instance level and when it exceeds the global config, the partitions with non-empty buffers would be paused.
    - **Pros**
        1. Simplest in terms of implementation.
    - **Cons**
        1. The issue with this scheme could be achieving fairness across Stream Threads. Consider the following scenario: Let's say there are more than 1 Stream Threads in an instance, and only one of them is exceeding the bounds individually. Since the algorithm is looking at the overall bytes count across all tasks within the instance, the other Threads would also end up paying the penalty and be paused. Note that, fairness across Stream Tasks within a Stream Thread is not a problem in any scenario as the consumer does round robin fetching.
- Fine grained memory allocation across Tasks/Partitions. The current proposed config does memory allocation only at a Stream Thread level and stops. One of the alternatives was to assign memory not just at a Thread level but trickling it down further at a Task level and then finally at a Partition level.
    - **Pros**
        - Assigning memory down to the Partition level would make the behaviour closest to the current **buffered.records.per.partition.**
        - We would pause only the exceeding partition instead of the proposed scheme where all non empty partitions are paused, which might lead to unwanted pauses.
    - **Cons**
        - The only reason this option was rejected was to keep things simple.
- Heuristic Based Pauses: Instead of pausing all non empty partitions, use some heuristics to pause the partitions like pause partitions accounting for X % of bytes or pick the one with most bytes and pause only those. This option was also rejected in favour of simplicity.