KIP-63: Unify store and downstream caching in streams

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *here*

Released: 0.10.1.0

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

Motivation

Today a stateful processor node, such as one that performs aggregates, stores intermediate data in a local state store as well as forwards it downstream to the next processor node. Local stores usually have a cache to batch writes and reduce the load on their backend. However, no such cache exists for data sent downstream. This increases both the CPU load on the system as well as the load on Kafka itself where data is ultimately stored.

Example:

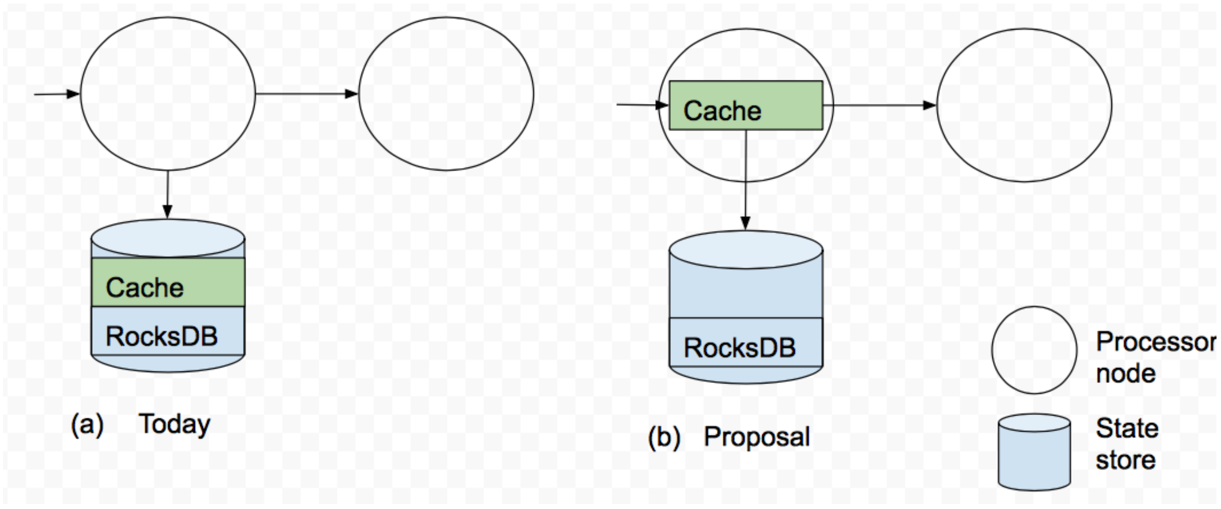
- The input is a sequence of messages <K,V>: <K1, V1>, <K2, V5>, …, <K1, V10>, <K1, V100> (Note: The focus in this example is on the messages with key == K1.)
- A processor node computes the sum of values, grouped by key, for the input above.
- In the current implementation, what is emitted for key K1 is a sequence of Change Values: <V1, null>, <V1 + V10, V1>, <V1 + V10 + V100, V1 + V10>

- That is, 3 write operations in this particular example.

These 3 writes do not not cause much load on the local RocksDB store because they are cached in-memory first, and only the final write would be written to RocksDB. However, the output from all 3 writes is forwarded to downstream processor nodes.

Proposed Changes

We propose a unified cache for both state store as well as sending downstream to the next processor, as illustrated in the Figure below:



The cache has two functions. First, it continues to serve as a read buffer for data that is sent to the state store, just like today. Second, it serves as a write deduplicator for the state store (just like today) as well as for the downstream processor node(s). So the tradeoff is "getting each update, i.e., a low update delay -- but a large CPU and storage load" vs. "skipping some updates, i.e., larger update delay -- but a reduced resource load". Note that this optimization does not affect correctness. The optimization is applicable to aggregations. It is not applicable to other operators like joins.

The proposal calls for one parameter to be added to the streams configuration options:

1. `cache.max.bytes.buffering`: This parameter controls the **global** cache size. Specifically, for a streams instance with T threads and cache size C, each thread will have an even C/T bytes of cache, to use as it sees fit among its tasks. No sharing of caches across threads will happen. Note that the cache serves for reads and writes. Records are evicted using a simple LRU scheme once the cache size is reached. The first time a keyed record R1 = <K1, V1> finishes processing at a node, it is marked as dirty in the cache. Any other keyed record R2 = <K1, V2> with the same key K1 that is processed on that node during that time will overwrite <K1, V1>. Upon flushing R2 is i) forwarded to the next processing node and ii) written to RocksDB (one write is local, one is to a backing Kafka topic).

The semantics of this parameter is that data is forwarded and flushed whenever the earliest of commit.interval.ms (note this parameter already exists and specifies the frequency with which a processor flushes its state) or cache pressure hits. These are global parameters in the sense that they apply to all processor nodes in the topology, i.e., it will not be possible to specify different parameters for each node. Such fine grained control is probably not needed from the DSL.

Continuing with the motivating example above, what is forwarded for K1, assuming all three operations hit in cache, is a final Change value <V1 + V10 + V100, null>.

For the low-level Processor API, we propose no changes, i.e., this KIP does not apply to the Processor API, just the DSL.

Public Interfaces

This KIP adds the `cache.max.bytes.buffering` configuration to the streams configuration as described above.

Proposed implementation outline

1. Change the RocksDB implementation for KStream windowed aggregations, to merge range queries from the cache with range queries from RocksDb.
2. Extract the LRU cache out of RocksDBStore, as a separate store for KGroupedStream.aggregate() / reduce(), and KStream.aggregateByKey() / reduceByKey(). Forward entires downstream on flush or evict.
3. Adjust context forwarding logic so that forwarding happens on cache flush
4. Add the above config into StreamsConfig

Compatibility, Deprecation, and Migration Plan

- *The correctness of the streams application is not affected for existing users who have not set this parameter. Users might notice that records are not forwarded downstream immediately but after a certain time.*

Alternatives and future works

- **Add user-facing triggers based on number of messages**. We considered adding the notion of count triggers to the DSL, where a user would explicitly set up a trigger to forward messages downstream after a certain number of messages have been processed, e.g., see CountTrigger below:

```
KTable<Windowed<String>, Long> wordCounts = textLine

   .flatMapValues(value ->Arrays.asList(value.toLowerCase().split("\\W+")))

   .map((key, word) -> new KeyValue<>(word, word)

   .countByKey(TimeWindows.of("WindowName", 10 * 1000L)

   .trigger(new CountTrigger(10));
```

- **Add user-facing triggers based on time**. We considered adding the notion of time triggers to the DSL, where a user would explicitly set up a trigger to forward messages downstream after a certain stream time has passed, e.g., see StreamTimeTrigger below:

```
KTable<Windowed<String>, Long> wordCounts = textLine

   .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))

   .map((key, word) -> new KeyValue<>(word, word)

   .countByKey(TimeWindows.of("WindowName", 10 * 1000L)

   .trigger(new StreamTimeTrigger(5 * 1000L));
```

There are several drawbacks with explicit triggers. First, we believe that for the majority of use cases, we improve the user experience by reducing resource usage with no complexity in the DSL. Second, explicit triggers force the user to hardcode in their application the tradeoff between latency and data volume. This proposal argues that is best done in a separate config file, appropriate for each deployment environment. Of course, a strength of explicit triggers is that some users might want to have explicit finer-grained control.

- **Output when end time for a window is reached + for late arriving records**. For windows-based aggregates, we considered a form of implicit triggers based on the end time for a window. Records would be forwarded downstream only when the window end time is reached. In addition, any late arriving records would be immediately forwarded.

The drawback of this approach is that window sizes are rather arbitrary and may not provide a good tradeoff between latency and resource usage. For example, if a window is 5 minutes in length, the user cannot specify latencies smaller than 5 minutes. This option would still require us to bring the store cache to the processor so that the RocksDb load could also be controlled. Hence, the actual work needed is the same as for our proposal.

- **Add another parameter `cache.max.write.buffering.ms`**: The first time a keyed record R1 = <K1, V1> finishes processing at a node, it is assigned a timestamp T1. R1 waits at most for the configured amount of (wall-clock) time above before being forwarded downstream. Any other keyed record R2 = <K1, V2> with the same key K1 that is processed on that node during that time will overwrite <K1, V1>. R2 inherits T1 on overwrite. If (current time >= T1 + the above config timeout) then R2 is i) forwarded to the next processing node and ii) written to RocksDb.

The advantage of this parameter is that it could be smaller than the existing `commit.interval.ms,` and thus allow for smaller time intervals spent buffering. However, if desired, `commit.interval.ms` could also be set to a smaller value, so it is unclear what the benefit of yet another additional timeout would be.

- **Expose this caching to the Processor API:**

For the low-level Processor API, we could allow users to enable/disabling caching on a store-by-store basis using a new .enableCaching() call. For example:

```
TopologyBuilder builder = new TopologyBuilder();


builder.addStateStore(Stores.create("Counts").withStringKeys().withIntegerValues().persistent().enableCaching().build(), "Process");
```

The implication of adding caching to the processor API is that the caching layer will also do forwarding. Hence if a store's cache is enabled, the use of context.forward would not needed anymore. However, this poses the problem of a user having to change their code (to use context.forward) each time they want to enable or disable caching. As such, more thought is needed for this approach. Furthermore, there are several other ways of doing caching in the processor API and perhaps we do not want to prescribe this one as the only way.