KIP-138: Change punctuate semantics

- Status
- Motivation
- Public Interfaces
- Proposed Changes
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives

Status

Current state: Accepted

Discussion thread: *here*

JIRA: *KAFKA-5233*

Released: 1.0.0

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

Motivation

Currently punctuate is triggered by the advance of the task's timestamp, which is the minimum of the timestamps of all input partitions. By default this means the event-time from the messages but a custom TimestampExtractor can be provided to use system-time instead of event-time. However, in that case the triggering of punctuate is still driven by the arrival of messages to all partitions and not by the advance of the system-time itself. The effect is that if any one of the input partitions has messages arriving irregularly, punctuate will be also be called at irregular intervals and in the extreme case not called at all if any one of the input partitions doesn't receive any messages.

Public Interfaces

`org.apache.kafka.streams.processor.Processor`

`org.apache.kafka.streams.processor.ProcessorContext`

`org.apache.kafka.streams.kstream.Transformer`

`org.apache.kafka.streams.kstream.ValueTransformer`

Terminology

Term	Description
Stream partition time	The value returned by the TimestampExtractor implementation in use or -1 if there haven't been any messages received for that partition. This can be the record timestamp, wall-clock time or any other notion of time defined by the user. However, as per the API doc, the extracted timestamp MUST represent the milliseconds since midnight, January 1, 1970 UTC. Please note that currently the TimestampExtractor is global to the KafkaStreams instance but after KIP-123 the extractor will be per source allowing multiple different extractors within a topology.
Stream time	Defined as the smallest among all its input stream partition timestamps (-1 if any of the partition hasn't received messages)
Punctuate time	Reference time used to trigger the Punctuate calls, currently the stream time.

Punctuate's timestamp argument	Currently the stream time when this method is being called
Punctuate's output record time	Record timestamp for records returned by Transformer.punctuate or generated from punctuate via ProcessorContext.forward. Currently the stream time.

Proposed Changes

The proposal is to deprecate the current punctuate() method on Processor, Transformer and ValueTransformer interfaces:

```
@Deprecated
void punctuate(long timestamp); // current
```

Add a new Punctuator functional interface:

Punctuator

```
interface Punctuator {
    void punctuate(long timestamp);
}
```

On ProcessorContext deprecate the current schedule method and add a new overload taking the Punctuator added:

ProcessorContext

```
@Deprecated
void schedule(long interval); //current, stream-time semantics

Cancellable schedule(long interval, PunctuationType type, Punctuator callback); //new
// We could allow this to be called once for each value of PunctuationType to mix approaches.
```

Where PunctuationType is

PunctuationType

```
enum PunctuationType {
  STREAM_TIME,
  WALL_CLOCK_TIME,
}
```

And Cancellable is

Cancellable

```
interface Cancellable {
  void cancel();
}
```

Cancellable return type is provided to cater for more complicated use cases as such described in the Punctuate Use Cases sub page. For those cases requiring stream-time based punctuation with a system-time upper bound (aka "hybrid" punctuation semantics) the following pattern can be used:

```
ProcessorContext context;
long streamTimeInterval = ...;
long systemTimeUpperBound = ...; //e.g. systemTimeUpperBound = streamTimeInterval + some tolerance
Cancellable streamTimeSchedule;
Cancellable systemTimeSchedule;

public void init(ProcessorContext context){
        this.context = context;
        streamTimeSchedule = context.schedule(PunctuationType.STREAM_TIME, streamTimeInterval,   this::
streamTimePunctuate);
        systemTimeSchedule = context.schedule(PunctuationType.WALL_CLOCK_TIME, systemTimeUpperBound, this::
systemTimePunctuate);
}

public void streamTimePunctuate(long streamTime){
        periodicBusiness(streamTime);

        systemTimeSchedule.cancel();
        systemTimeSchedule = context.schedule(PunctuationType.WALL_CLOCK_TIME, systemTimeUpperBound, this::
systemTimePunctuate);
}

public void systemTimePunctuate(long systemTime){
        periodicBusiness(context.timestamp());

        streamTimeSchedule.cancel();
        streamTimeSchedule = context.schedule(PunctuationType.STREAM_TIME, streamTimeInterval, this::
streamTimePunctuate);
}

public void periodicBusiness(long streamTime){
        // guard against streamTime == -1, easy enough.
        // if you need system time instead, just use System.currentTimeMillis()

        // do something businessy here
}
```

Compatibility, Deprecation, and Migration Plan

The following methods will be deprecated

- Processor.punctuate(long timestamp),
- Transformer.punctuate(long timestamp),
- ValueTransformer.punctuate(long timestamp),
- ProcessorContext.schedule(long interval);

The deprecated methods shall remain for some time along the newly added ones to allow for a smooth migration.

The semantics of the deprecated methods shall remain unchanged.

A call to the deprecated ProcessorContext.schedule(interval) from within a Processor will be equivalent to:

```
context.schedule(interval, PunctuationType.STREAM_TIME, this::punctuate);
```

A call to the deprecated ProcessorContext.schedule(interval) from within a Transformer will be equivalent to:

```
context.schedule(interval, PunctuationType.STREAM_TIME, timestamp -> {
        KeyValue<K,V> pair = punctuate(timestamp);
        if (record != null) {
                context.forward(pair.key, pair.value);
        }
});
```

Test Plan

Stream time, system time and a mix of both PunctuationTypes should be tested. Existing test for punctuation can be re-used to guide the test cases for stream time only. System time and mixed stream & system time tests will have to be developed.

Rejected Alternatives

(A) Change the semantics of `punctuate()` to be purely "system time driven", instead of "part time driven, and part data-driven". That is, the punctuate function triggering will no longer be dependent whether there are new data arriving, and hence not on the timestamps of the arriving data either. Instead it will be triggered only by system wall-clock time.

As for users, the programming pattern would be:

1. If you need to add a pure time-driven computation logic, use `punctuate()`.
2. If you need to add a data-driven computation logic, you should always use `process()`, and in `process()` users can choose to trigger some specific logic only every some time units, but still when a new data has arrived and hence being processed. With this a punctuation with semantics close to current ones can be achieved but giving user control over the details, as follows:

```
    long lastPunctuationTime = 0;

    long interval = <some-number>; //millis

    @Override
    public void process(K key, V value){

        while (ctx.timestamp() >= lastPunctuationTime + interval){

            punctuate(ctx.timestamp()); //trigger punctuate or any other method at current record
timestamp or lastPunctuationTime + interval, if the user prefers

            lastPunctuationTime += interval; // or do lastPunctuationTime = ctx.timestamp() if the user
prefers
        }

        // do some other business logic here

    }
```

Drawbacks:

- The above approach changes the semantics of the punctuate method and therefore is not backward-compatible.
- It is not clear if doing data-driven periodic operations from the process() method without the intricate calculations of minimum timestamp per input partition is sufficient to cater for all use cases that may be attainable using present day stream-time based punctuate

(B) An alternative could be to leave current semantics as the defaults for the punctuate method but allow a configuration switch between event and system time.

Drawback:

- It's reasonable to assume different semantics be needed in different parts of a topology and configuration is global to a KafkStreams instance, therefore this seems to be too limiting.

(C) Another alternative would be to leave current semantics as-is and add another callback method to the Processor interface that can be scheduled similarly to punctuate() but would always be called at fixed, wall clock based intervals

Drawback:

- This is similar to what's being proposed, however, the functional interface approach offers more flexibility in that the same lambda/method reference can be passed as a parameter to ProcessorContext.schedule() as a callback for both PunctuationTypes.

(D) Yet another alternative would be to leave current semantics as-is but allow users to provide a function determining the timestamp of the stream task. In a similar way to how the TimestampExtractor allows users to decide what the current timestamp is for a given message (event-time, system-time or other), this feature would allow users to decide what the current timestamp is for a given stream task irrespective of the arrival of messages to all of the input partitions. This approach brings more flexibility at the expense of added complexity.

Drawback:

- The scope of this KIP is to re-define punctuate semantics only, without alterations to the notions of stream-time itself, which the alternative would require.

(E) Finally, the hybrid approach (this is convenient for the use cases in Punctuate Use Cases):

ProcessorContext

```
/**
* Schedule punctuate at stream-time intervals with a system-time upper bound.
* For pure system-time based punctuation streamTimeInterval can be set to -1 == infinite
* and systemTimeUpperBound to the desired interval
*/
schedule(Punctuator callback, long streamTimeInterval, long systemTimeUpperBound);

/**
* Schedule punctuate at stream-time intervals without a system-time upper bound,
* i.e. pure stream-time based punctuation
*/
schedule(Punctuator callback, long streamTimeInterval);
```

Punctuation is triggered when either:

- the stream time advances past the (stream time of the previous punctuation) + streamTimeInterval;
- or (iff systemTimeUpperBound is set) when the system time advances past the (system time of the previous punctuation) + systemTimeUpperBound

In either case:

- we trigger punctuate passing as the argument the stream time at which the current punctuation was meant to happen
- next punctuate is scheduled at (stream time at which the current punctuation was meant to happen) + streamTimeInterval

Drawbacks:

- It's been argued this type of hybrid punctuation is more difficult to reason about than separate stream-time and system-time punctuations and the approach need further thought
- Some problems with this algorithm have been identified for edge case scenarios (see discussion thread)
- The various trade-offs of this approach may better be left to the the users as per the mantra "make simple thing easy and complex things possible"

The current proposal opens the door to adding more PunctuationTypes in the future and so after further discussion and in a separate KIP, other approaches such as the hybrid one can be added later on.

However, hybrid semantics can be implemented on top of the 2 PunctuationType callbacks, as show in the Proposed Changes section. This gives users more flexibility in addressing the various trade-offs inherent in this design as is most appropriate to their use case.