# FLIP-228: Support Within between events in CEP Pattern

## Status

| | |
|---|---|
| **Discussion thread** | |
| **Vote thread** | |
| **JIRA** | |
| **Release** | |

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

## Motivation

Pattern#withIn interface defines the maximum time interval in which a matching pattern has to be completed in order to be considered valid, which interval only corresponds to the maximum time gap between first and the last event. The interval representing the maximum time gap between the previous and current event is required to define in the certain scenario, for example, purchases good within a maximum of 5 minutes after browsing. Within between events is proposed in Pattern to support the definition of the maximum time interval in which a completed matching pattern is considered valid, which interval represents the maximum time gap between events for matching Pattern.

## Public Interfaces

The following public interface is introduced to support the within between events requirement.

### Adds Pattern#within(WithinType withInType, Time windowTime)

Pattern#within(WithinType withinType, Time windowTime) is introduced to define the maximum time interval in which a matching pattern has to be completed in order to be considered valid. The time interval corresponds to the maximum time gap between events like between the previous and current event or between the first and last event. Pattern#within(WithinType withinType, Time windowTime) is used to configure the maximum time interval between events and return the same pattern operator with the specified new window length that throws the MalformedPatternException if window length between the previous and current event is larger than which between the first and last event.

The time interval of Pattern#within(WithinType.PREVIOUS_AND_CURRENT, windowTime) is meant that for example there is a pattern like Pattern.<Event>begin("p1").where(...).followBy("p2").where(...).within(WithinType.PREVIOUS_AND_CURRENT, Time.minutes(1)).where(...).followBy("p3").where(...).within(WithInType.PREVIOUS_AND_CURRENT, Time.minutes(2)). The pattern is designed to find the event sequences which are p1->p2->p3, the time interval between p1 and p2 matching to the event is within 1 minute, and the time interval between p2 and p3 matching to the event is within 2 minute.

**Pattern#within(WithinType withinType, Time windowTime)**

```
/**
  * Defines the maximum time interval in which a matching pattern has to be completed in order to
  * be considered valid. This interval corresponds to the maximum time gap between events.
  *
  * @param withinType Type of the within interval between events
  * @param windowTime Time of the matching window
  * @return The same pattern operator with the new window length
  * @throws MalformedPatternException if window length between the previous and current event is larger
  *                                   than which between the first and last event.
  */
public Pattern<T, F> within(WithinType withinType, Time windowTime) {...}
```

**WithinType**

```
/**
  * Type enum of time interval corresponds to the maximum time gap between events.
  */
public enum WithinType {
    // Interval corresponds to the maximum time gap between the previous and current event.
    PREVIOUS_AND_CURRENT,
    // Interval corresponds to the maximum time gap between the first and last event.
    FIRST_AND_LAST;
}
```

## Adds Pattern#times(int times, Map<Integer, Long> windowTimes)

Pattern#times(int times, Map<Integer, Long> windowTimes) is introduced to specify exact number of times that this pattern should be matched and time interval corresponds to the maximum time gap between previous and current event for each times. Pattern#times(int times, Map<Integer, Long> windowTimes) is used to configure the mapping between number of times matching event must appear and time of the matching window, which allows users to define the maximum time interval for each times.

**Pattern#times(int times, Map<Integer, Long> windowTimes)**

```
/**
  * Specifies exact number of times that this pattern should be matched and time interval
  * corresponds to the maximum time gap between previous and current event for each times.
  *
  * @param times number of times matching event must appear
  * @param windowTime time of the matching window between times
   * @return The same pattern with number of times applied
  * @throws MalformedPatternException if the quantifier is not applicable to this pattern.
  */
public Pattern<T, F> times(int times, Time windowTime) {...}
```

## Adds Pattern#times(int from, int to, Map<Integer, Long> windowTimes)

Pattern#times(int from, int to, Map<Integer, Long> windowTimes) is introduced to specify that the pattern can occur between from and to times with time interval corresponds to the maximum time gap between previous and current event for each times. Pattern#times(int from, int to, Map<Integer, Long> windowTimes) is also used to configure the mapping between number of matching times and time of the matching window.

**Pattern#times(int from, int to, Map<Integer, Long> windowTimes)**

```
/**
  * Specifies that the pattern can occur between from and to times with time interval corresponds
  * to the maximum time gap between previous and current event for each times.
  *
  * @param from number of times matching event must appear at least
  * @param to number of times matching event must appear at most
  * @param windowTime time of the matching window between times
  * @return The same pattern with the number of times range applied
  * @throws MalformedPatternException if the quantifier is not applicable to this pattern.
  */
public Pattern<T, F> times(int from, int to, Time windowTime) {...}
```

## Adds Pattern#timesOrMore(int times, Map<Integer, Long> windowTimes)

Pattern#timesOrMore(int times, Map<Integer, Long> windowTimes) is introduced to specify that this pattern can occur the specified times at least with interval corresponds to the maximum time gap between previous and current event for each times, which means at least the specified times and at most infinite number of events can be matched to this pattern. Pattern#timesOrMore(int times, Map<Integer, Long> windowTimes) is also used to configure the mapping between at least the specified times and time of the matching window.

**Pattern#timesOrMore(int times, Map<Integer, Long> windowTimes)**

```
/**
  * Specifies that this pattern can occur the specified times at least with interval corresponds
  * to the maximum time gap between previous and current event for each times. This means at least
  * the specified times and at most infinite number of events can be matched to this pattern.
  *
  * @param times number of times at least matching event must appear
  * @param windowTime time of the matching window between times
  * @return The same pattern with a {@link Quantifier#looping(ConsumingStrategy)} quantifier
  *     applied.
  * @throws MalformedPatternException if the quantifier is not applicable to this pattern.
  */
public Pattern<T, F> timesOrMore(int times, Time windowTime) {...}
```

## Adds Pattern#oneOrMore(int times, Map<Integer, Long> windowTimes)

Pattern#oneOrMore(Map<Integer, Long> windowTimes) is introduced to specify that this pattern can occur one or more times and time interval corresponds to the maximum time gap between previous and current event for each times, which means at least one and at most infinite number of events can be matched to this pattern. Pattern#oneOrMore(Map<Integer, Long> windowTimes) is also used to configure the mapping between at least one times and time of the matching window.

**Pattern#oneOrMore(Map<Integer, Long> windowTimes)**

```
/**
  * Specifies that this pattern can occur {@code one or more} times and time interval corresponds
  * to the maximum time gap between previous and current event for each times. This means at least
  * one and at most infinite number of events can be matched to this pattern.
  *
  * <p>If this quantifier is enabled for a pattern {@code A.oneOrMore().followedBy(B)} and a
  * sequence of events {@code A1 A2 B} appears, this will generate patterns: {@code A1 B} and
  * {@code A1 A2 B}. See also {@link #allowCombinations()}.
  *
  * @param windowTime time of the matching window between times
  * @return The same pattern with a {@link Quantifier#looping(ConsumingStrategy)} quantifier
  *     applied.
  * @throws MalformedPatternException if the quantifier is not applicable to this pattern.
  */
public Pattern<T, F> oneOrMore(Time windowTime) {...}
```

# Example Usage

Example section provides code snippets that shows how user invokes Pattern#within(withinType, windowTime) interface to define the maximum time interval between the previous and current event. For example, in the user behavior analysis scenario, event sequences that a purchase event occurs within a maximum of 5 minute after browsing and a pay event occurs within a maximum of 3 minute after purchasing are required to statistics and output the timeout event sequences. The business developer formulates the following Pattern that determines the statistics requirement:

**Pattern using within(WithinType.PREVIOUS_AND_CURRENT, windowTime)**

```
Pattern pattern = Pattern.<Event>begin("start").where(new SimpleCondition<Event>() {
  @Override
  public boolean filter(Event value) throws Exception {
    return value.getName().equals("browse");
  }
})
.followedBy("middle").where(new SimpleCondition<Event>() {
  @Override
  public boolean filter(Event value) throws Exception {
    return value.getName().equals("purchase");
  }
}).within(WithinType.PREVIOUS_AND_CURRENT, Time.minutes(5))
.followedBy("end").where(new SimpleCondition<Event>() {
  @Override
  public boolean filter(Event value) throws Exception {
    return value.getName().equals("pay");
  }
}).within(WithinType.PREVIOUS_AND_CURRENT, Time.minutes(3));
```

Whenever a pattern has a window length attached via the within(WithinType.PREVIOUS_AND_CURRENT, windowTime) method, it is possible that partial event sequences are discarded because they exceed the window length. To act upon a timed out partial match one can use TimedOutPartialMatchHandler interface supposed to be used in a mixin style. The TimedOutPartialMatchHandler provides the additional processTimedOutMatch method which will be called for every timed out partial match.

**TimedOutPartialMatchHandler implementation**

```
public class TimedOutPatternProcessFunction<IN, OUT> extends PatternProcessFunction<IN, OUT> implements
TimedOutPartialMatchHandler<IN> {

    private final OutputTag<String> outputTag;

    public TimedOutPatternProcessFunction(OutputTag<String> outputTag) {
        this.outputTag = outputTag;
    }

    @Override
    public void processMatch(Map<String, List<IN>> match, Context ctx, Collector<OUT> out) throws Exception;
        IN startEvent = match.get("start").get(0);
        IN middileEvent = match.get("middile").get(0);
        IN endEvent = match.get("middile").get(0);
        out.collect(OUT(startEvent, middileEvent, endEvent));
    }

    @Override
    public void processTimedOutMatch(Map<String, List<IN>> match, Context ctx) throws Exception;
        IN startEvent = match.get("start").get(0);
        ctx.output(outputTag, T(startEvent));
    }
}
```

Using the above Pattern and TimedOutPartialMatchHandler, the developer creates the PatternStream with the pattern and the DataStream from the created PatternStream with the input Event Datastream and the process function TimedOutPatternProcessFunction.

**Analysis DataStream example**

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern);
OutputTag<String> outputTag = new OutputTag<String>("timeout-output"){};
SingleOutputStreamOperator<ComplexEvent> analysisResult = patternStream.process(
    new TimedOutPatternProcessFunction(outputTag));
DataStream<TimeoutEvent> timeoutResult = analysisResult.getSideOutput(outputTag);
analysisResult.print();
timeoutResult.print();
```

The developer submits the analysis DataStream application to execute above user behavior analysis. The event sequences that purchase behavior occurs within a maximum of 5 minute after browsing and pay behavior occurs within a maximum of 3 minute after purchasing, and the timeout sequences are output in terminal.

# Proposed Changes

## Introduces windowTimes in NFA

In order to achive the functional requirements goal of within(WithinType.PREVIOUS_AND_CURRENT, windowTime) invoker, windowTimes is introduced as class member in NFA to define the length of a windowed part for Pattern, which represents the mapping between the name of valid NFA state and corresponding window length, as specified using the Pattern#partialWithin(Time) method. Different from the windowTime class member that determines the timeout from the start to the end of the valid NFA states, the windowTimes only determines timeout of each individual NFA state.

From the perspective of compilation, NFAFactoryCompiler builds the windowTimes that is mapping between the name and corresponding window length of valid NFA states while createing all the states between Start and Final state, and passes the window length mapping when compiling a Pattern into a NFA. In terms of NFA execution, when computing the next computation states based on the given computation state, the current event, its timestamp and the internal state machine, stateTimestamp is introduced in ComputationState to represent the timestamp of the last element in the current computation state. If the timeout condition of individual NFA state is met, in other words, the value of the timestamp of the current event minus the stateTimestamp of the current computation state exceeds the corresponding windowTime, NFA clears the sharedBuffer and also emits all timed out partial matches of the individual state when CepOperator advances the time for the given NFA to the given timestamp.

# Compatibility, Deprecation, and Migration Plan

No.

# Test Plan

Unit tests of Pattern#within(withinType, windowTime) will be added in flink-cep module to cover the correctness of Pattern including within between events.

# Rejected Alternatives

No.