PredicateWindow

Remark: This window is **very complex and may somethime not behave as expected**. It can simulate any other window, but if not necessary, you should prefer TimeWindow or ElementWindow

The predicate window opens and closes the window regarding a start and optional an end condition. Elements that are not inside a window are discarded and send to output port 1.

The operator works as follows:

- It **first** checks, if the `maxWindowTime` is reached. In this case all internal buffers for each group are cleared (and the content is send to the output when `outputIfMaxWindowTime` is true), where the first element is older than the given threshold.
- After then, it checks, if `closeWindowAfterNoupdatesFor` is set and closes all buffers where the last element has reached the buffer a time longer than the parameter.
- The operator determines the group (`partition`) for the current input.

- After that, if set, the **`clear`** condition is checked for the current group and the current input.

- If the window for this group **is already opened** (this means, that **some elements before** the **start predicates has been evaluated to true**) the next step is to check,
  - if the **end condition** is true. Then the operator creates an **output**. Typically, the whole window is written and the buffer is cleared. With the `clear` and `advanceWhen` condition, this behaviour can be changed.
  - if the **end condition** is false, the current element is added to the window and kept inside the operator.
- If the window for this group **is not opened**, the `start` condition is checked. It the condition is true, the operator **opens a new window** and adds the current element to the window.
  - the end predicate is ignored in this case, you could use `allowSameStartAndEndTS` to allow single element windows! Attention, in this case, the produced result is not really valid (as start and end timestamp are the same) and you need to do some window processing afterwards.

For the output there are different configurations:

- `samestarttime`: Each element in the output will get the same starttime, i.e. the starttime from the first element. This could be used in Aggregate (and Group) operator to get only a single result for a complete window.
- `nesting`: In Odysseus the output is typically a set of elements that are send one after the other. If this flag is set to true, the output of the window is a single list, with all elements from the window. This can be advantage, if the processing afterwards treats the elements together (e.g. in a MAP-Operation). `Samestarttime` sets the time for each element in the list, the list get the union of all intervals inside the list.

To simulate some kind of sliding window, the following parameters can be used:

- `AdvanceWhen`: This condition checks, if the window should move, i.e. if elements in the current buffer (for the current group) should be removed. If this predicate evalutes to true, the next parameter is used to determine which number of elements the move of the window should be
- `AdvanceSize`: This size tells the operator if cases of `AdvanceWhen` is true, how many elements should be removed from the start of the current window. If the value is below 0 or the current window has less elements that this value, the buffer is cleared.

Remark: Advance is only used, if an output is generated and will be used after the results are produced. To clear the buffer independent of an output, clear needs to be used.

Parameters

- `start:` The start condition for a predicate window. If the condition evaluates to true, the windows is opened until the end predicate evaluates to true (or if not given the start predicate evaluates to false). Note, that all elements that are not inside a window are send to ouput port 1
- `end:` The end condition for a predicate window. The tuple for which this condition is evaluated to true is only part of the result, if `keepEndingElement` is set to true!
- `clear`: If this parameter is set, the window will only be cleared, if the **condition** is true. By this, the same element can be part of multiple windows (sliding)
- `sameStartTime`: If set to true, all produced elements get the same start timestamp
- `size:` The maximum size of the window. Can be either a single number or a pair of a number and a time unit. Possible values for the unit are one of TimeUnit like SECONDS, NANOSECODS etc. - default time is the base time of the stream (typically milliseconds)
- `keepEndingElement`: Typically, the object that fulfils the end condition will not be part of the result. If setting this attribute to true, the element will be part
- `partition`: Evaluate the predicates on partitioned defined by different values of this attribute (similar to group by in aggreation)
- `useElementOnlyForStartOrEnd`: Typically, an object is used to evaluate **the start and the end condition** and an element can be used **for both** and can be part of multiple windows, i.e. the same element can be used to close a window and open a new window. If set to **true**, **only the start or end predicate** will be evaluted, i.e. an element **cannot** be used to close an open window and use the same element to open a new window.
- `keepTimeOrder`: If set to false, the ouput generation does not care about order. Typically, this makes only sense when using **`nesting=true!`**
- `closeWindowWithHeartbeat:` if true, the window is closed when a heartbeat is received. Take a look at the session window to see how it works.
- `closeWindowAfterNoupdatesFor`: A time parameter by which the window could be closed if some time no new element reaches the buffer. Mostly makes sense for partioned windows but works also with heartbeats.
- `closeWindowAfterNoUpdateTimePort`: In cases, the window closes because of `closeWindowAfterNoUpdateTime` this port is used for the output. Default is 0, i.e. the default output port. This can be used to handle outputs of this kind differently.
- `nesting`: Default false. If set to true, elements that are grouped together are written in a single object as list.

Parameters for MaxWindowTime

- `maxWindowTime:` The maximum possible age of a window. If reached, the current window is closed.
- `outputIfMaxWindowTime`: A window can close by condition or when maxWindowTime is reached. Set to false to avoid writing in case of maxWindowTime (default is true)
- `maxWindowTimeOutputPort`: A special output port can be defined to allow to write in cases where maxWindowTime is reached to this port. Default is 0, i.e. the default output port.

Remark: This is a blocking operator. The operator does not write elements before it sees new elements not belonging to the window anymore (similiar to ElementWindow)

Example

In the following we provide some examples and the corresponding output.

As input, we assume the following simple input:

```
ID        Time        isLast
A    1    false
A    2    false
A    3    false
A    4    true
B    5    false
B    6    false
B    7    false
B    8    false
B    9    false
B    10    true
C    11    false
C    12    false
C    13    false
C    14    false
C    15    false
C    16    false
```

Preprocessing

With some preprocessing (to allow more examples)

```
#PARSER PQL
#ADDQUERY
in = CSVFILESOURCE({SCHEMA = [['ID', 'String'],['pos','STARTTIMESTAMP'],['isLast','Boolean']], DELIMITER =
'\t', SOURCE = 'source', FILENAME = '${PROJECTPATH}/input.csv'})

map = STATEMAP({EXPRESSIONS = [['isNull(__last_1.ID) OR (__last_1.ID != ID)','newElem']], KEEPINPUT = true}, in)
```

we will get:

ID	pos	isLast	newElem	start	end
C	16	false	false	16	<null>
C	15	false	false	15	<null>
C	14	false	false	14	<null>
C	13	false	false	13	<null>
C	12	false	false	12	<null>
C	11	false	true	11	<null>
B	10	true	false	10	<null>
B	9	false	false	9	<null>
B	8	false	false	8	<null>
B	7	false	false	7	<null>
B	6	false	false	6	<null>
B	5	false	true	5	<null>
A	4	true	false	4	<null>
A	3	false	false	3	<null>
A	2	false	false	2	<null>
A	1	false	true	1	<null>

As you can see, the new field `newElem` is added, that is set to true, if it is **the first element** or if the element is **different that the last element**.

Using only a start predicate

```
win = PREDICATEWINDOW({start = 'newElem', SAMESTARTTIME = true}, map)
```

will result in:

ID	pos	isLast	newElem	start	end
C	11	false	true	11	12
B	5	false	true	5	6
A	1	false	true	1	2

Here the window is **opened for every true evaluation of the start condition** and is **closed for every evaluation of !start**. All elements between these elements are discarded. They do not open a new window. This may not be, what you expect!

By this, you could keep a window open **as long the start condition is true**.

Using an end predicate

```
win = PREDICATEWINDOW({start = 'true', end = 'newElem', SAMESTARTTIME = true}, map)
```

ID	pos	isLast	newElem	start	end
B	10	true	false	5	11
B	9	false	false	5	11
B	8	false	false	5	11
B	7	false	false	5	11
B	6	false	false	5	11
B	5	false	true	5	11
A	4	true	false	1	5
A	3	false	false	1	5
A	2	false	false	1	5
A	1	false	true	1	5

Here each time a new window opens, the old window is closed, i.e. the same input element is responsible for starting and closing a window. The output contains two windows, one from 1 to 5 and one from 5 to 11. **As the final window is not closed**, the out starting from C (at 11) is discarded. Use DrainAtDone = true, to allow the output of these elements.

Using a start and an end predicate and keeping the ending element:

```
win = PREDICATEWINDOW({start = 'newElem', end = 'isLast', KEEPENDINGELEMENT = true, SAMESTARTTIME = true}, map)
```

will result in:

ID	pos	isLast	newElem	start	end
B	10	true	false	5	10
B	9	false	false	5	10
B	8	false	false	5	10
B	7	false	false	5	10
B	6	false	false	5	10
B	5	false	true	5	10
A	4	true	false	1	4
A	3	false	false	1	4
A	2	false	false	1	4
A	1	false	true	1	4

Remark the difference: This operator blocks only until the end predicate is reached. This works only, if samestarttime is set to true, else e.g. A|4|true|false | META | 1|4 would be A|4|true|false | META | 4|4, this has no validitiy and will not be produced.

When you want to close the window with closing the stream, you could as always use DrainAtDone=true.

```
win = PREDICATEWINDOW({start = 'newElem', end = 'isLast', KEEPENDINGELEMENT = true, SAMESTARTTIME = true,
DRAINATDONE = true}, map)
```

ID	pos	isLast	newElem	start	end
C	16	false	false	11	17
C	15	false	false	11	17
C	14	false	false	11	17
C	13	false	false	11	17
C	12	false	false	11	17
C	11	false	true	11	17
B	10	true	false	5	10
B	9	false	false	5	10
B	8	false	false	5	10
B	7	false	false	5	10
B	6	false	false	5	10
B	5	false	true	5	10
A	4	true	false	1	4
A	3	false	false	1	4
A	2	false	false	1	4
A	1	false	true	1	4

Using predicate window to simulate standard windows

The predicate window can be used to simulate other windows. Remark: This is only for demonstration purposes as the element and time window are much faster!

All the following examples use this source:

```
#PARSER PQL
#RUNQUERY
timer = TIMER({
            period = 1000,
            source = 'timer'
          }
        )

ticker := MAP({
              expressions = [['counter()','tick'],['TimeInterval.START','TS']]
            },
            timer
          )
```

Element Window

```
#PARSER PQL
#IFSRCNDEF ticker
#INCLUDE ${PROJECTPATH}/TickerSource.qry
#ENDIF

#ADDQUERY
/// Tumbling element window
out = PREDICATEWINDOW({
                  /// Start window with every element
          start = "true",
          /// close window, when size of buffer is 5
          end = "size(__all)==5",
                  /// output as list
          nesting = true
        },
        ticker
      )
```

```
#PARSER PQL
#IFSRCNDEF ticker
#INCLUDE ${PROJECTPATH}/TickerSource.qry
#ENDIF

#ADDQUERY
/// Sliding element window
out = PREDICATEWINDOW({
                  /// start window with every element
          start = "true",
          /// If set to false, end element is not part of result
          KEEPENDINGELEMENT = true,
                  /// end predicate is tested before element is added to window
                  /// thats why size(__all) must be one below window size!
          end = "size(__all)=2",
          /// Move window when size of window is 3
          ADVANCEWHEN = 'size(__all)=3',
          /// move by one position
          ADVANCESIZE = 1,
          /// output as list
          nesting = true
        },
        ticker
      )
```

Time Window

```
#PARSER PQL
#IFSRCNDEF ticker
#INCLUDE ${PROJECTPATH}/TickerSource.qry
#ENDIF

#ADDQUERY
/// Tumbling time window
out = PREDICATEWINDOW({
                start = "true",
                end = "!isNull(__first.TS) && __first.TS + 10 < TS",
                nesting = true
        },
        ticker
      )
```