

Aggregation and Window

In this tutorial you will learn how to write aggregation queries and learn about the window concept in Odysseus.

We will use the same setting as in [Simple Query Processing](#). So you should follow steps 1-4.

Aggregation

Odysseus provides a wide range of aggregation operators. In this tutorial we will focus on the typical aggregations average (AVG), count (COUNT), sum (SUM), minimum (MIN) and maximum (MAX).

Create a new Odysseus Script file with the PQL template an name it aggregation1. Change #RUNQUERY to #ADDQUERY. By this, the query will only be installed and not started, so its easier to see what happens.

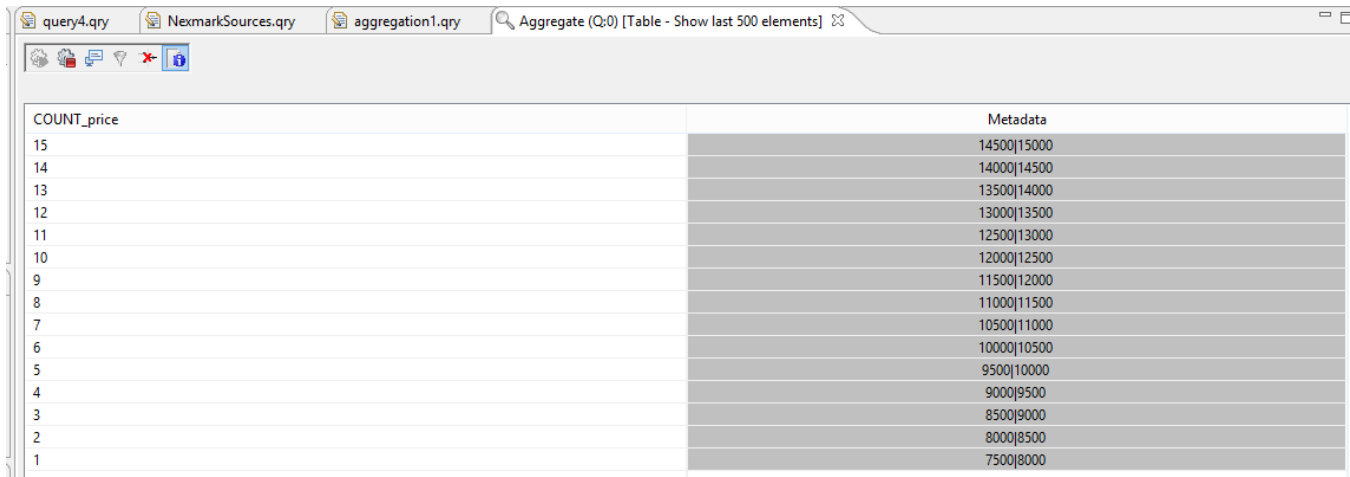
Write the following query:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = AGGREGATE({
    aggregations=[['COUNT', 'price', 'COUNT_price', 'integer']]
},
    nexmark:bid
)
```

Here you can see the aggregation operator that connects to source nexmark:bid. The aggregate functions are defined by the aggregations parameter. In this example you can see four entries. The first entry defines the aggregate function, COUNT in this case. The second entry, the attribute can should be used for the aggregation. For COUNT is does not matter, which attribute of the input element should be treated, but one attribute must be given, so be choose price. The third parameter defines the name of the output attribute, i.e. what is the new name for the output and finally, the last parameter determines the output data type. This parameter is optional.

Now execute the script, and show the result stream (e.g. as a table). Now start the query (e.g. with the selection of the Start Query Button).

The result should be something like this:



COUNT_price	Metadata
15	14500 15000
14	14000 14500
13	13500 14000
12	13000 13500
11	12500 13000
10	12000 12500
9	11500 12000
8	11000 11500
7	10500 11000
6	10000 10500
5	9500 10000
4	9000 9500
3	8500 9000
2	8000 8500
1	7500 8000

Here you can see 15 bids that have entered the system.

Other aggregations can be added easily, by adding another value to the aggregations parameter.

Remove the old query.

In the Project Explorer

- right click on the aggregation1.qry file and choose Copy
- right click on the Project Name (FirstSteps) and choose Paste
- Enter a new name for the query: aggregation2.qry
- Open the new file by double clicking in the project explorer

Replace the input with the following:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = AGGREGATE({
    aggregations=[
        ['COUNT', 'price', 'COUNT_price', 'integer'],
        ['AVG', 'price', 'AVG_price'],
        ['SUM', 'price', 'SUM_price'],
        ['MIN', 'price', 'MIN_price'],
        ['MAX', 'price', 'MAX_price']
    ]
},
    nexmark:bid
)
```

The results of the query should look somehow like the following:

COUNT_price	AVG_price	SUM_price	MIN_price	MAX_price	Metadata
17	177.2941176470588	3014.0	53.0	262.0	15500 16000
16	173.5625	2777.0	53.0	262.0	15000 15500
15	170.53333333333333	2558.0	53.0	262.0	14500 15000
14	167.64285714285714	2347.0	53.0	262.0	14000 14500
13	160.3846153846154	2085.0	53.0	248.0	13500 14000
12	153.08333333333334	1837.0	53.0	231.0	13000 13500
11	146.0	1606.0	53.0	221.0	12500 13000
10	138.5	1385.0	53.0	199.0	12000 12500
9	137.22222222222223	1235.0	53.0	199.0	11500 12000
8	136.75	1094.0	53.0	199.0	11000 11500
7	127.85714285714286	895.0	53.0	192.0	10500 11000
6	117.16666666666667	703.0	53.0	182.0	10000 10500
5	115.4	577.0	53.0	182.0	9500 10000
4	98.75	395.0	53.0	161.0	9000 9500
3	78.0	234.0	53.0	103.0	8500 9000
2	65.5	131.0	53.0	78.0	8000 8500
1	53.0	53.0	53.0	53.0	7500 8000

Now its time to highlight the Metadata part a little more.

In the last column you can see two values, separated by "|". The two values show the validity of the event. 7500|8000 e.g. means, that this event was valid in real time starting at 7500 (in Nexmark this means after 7.5 seconds) and ends **before** 8000, mathematical this is a right open interval: [7500,8000).

If you look at the values, you can see that they are not overlapping. This should be rather clear, because the count e.g. can not be 1 and 2 at the same time. The metadata show, that every 500 (in this case milliseconds) a new bid is send to the system and a new aggregation is build.

Another important think to note is, that a new aggregation will be calculated, when a new event arrives a the system. For this example it take 8 seconds before the first value is created. For streams with a low data rate the [AssureHeartbeat](#) operator can be used to produce values on a regular base.

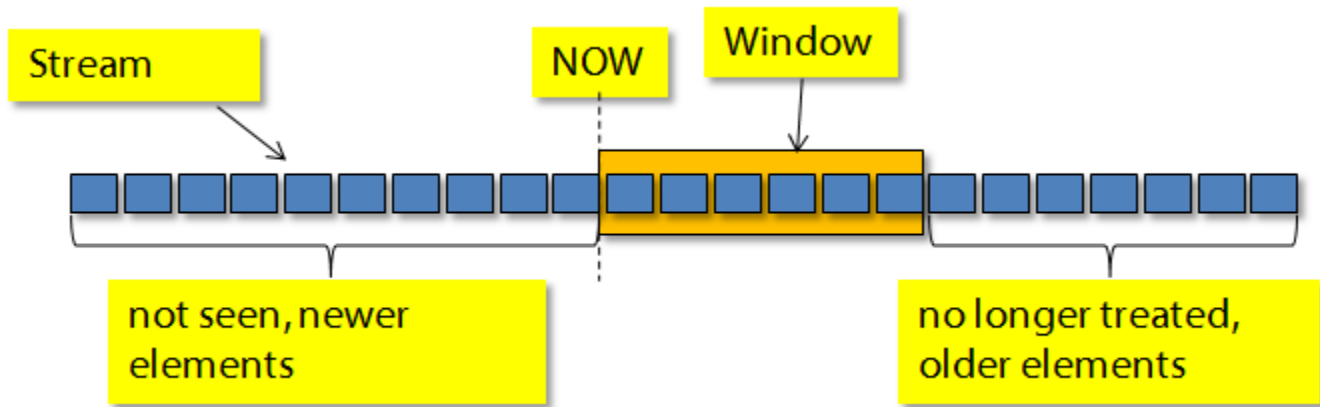
There is also the possibility to define groups (as in SQL group by). The below in [Element Based Windows](#) for an example.

Window

In the example above you can see that the aggregated values always contain the information of all previous events. Typically, the older the event is, the less interesting is it. An auction system provider may not only be interested in the alltime highest bit, but e.g. the highest bit for a specific time (e.g. the last 24 hours) or related to a number of bit that have been posed.

For this, Odysseus allows to define windows. A window defines a subset of elements that should be treated together, e.g. should be aggregated. These subset can overlap (so called sliding windows) or be disjunct (tumbling windows).

The following picture shows such a window. In this case 6 elements are in the window.



Odysseus provides three types of windows:

- Time based
- Element based
- Predicate based

Time Based Window

The time based window allows to define windows based on the time.

Create a new PQL script and name it timewindow1:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = TIMEWINDOW({SIZE=10000},nexmark:bid)
```

In this example we define a time based window over the bid source with a size of 10000, here 10 seconds.

If you run the query and look at the results you will see something like this:

timestamp	auktion	bidder	datetime	price	Metadata
20500	4	2	20501	158.0	20500 30500
20000	5	3	20001	76.0	20000 30000
19500	1	0	19500	296.0	19500 29500
19000	3	1	19001	257.0	19000 29000
18500	1	2	18501	273.0	18500 28500
18000	0	1	18001	162.0	18000 28000
17500	2	1	17500	236.0	17500 27500
17000	0	1	17001	151.0	17000 27000
16500	1	0	16500	267.0	16500 26500
16000	3	0	16001	241.0	16000 26000
15500	3	1	15500	237.0	15500 25500
15000	3	2	15001	219.0	15000 25000
14500	2	1	14500	211.0	14500 24500
14000	1	1	14001	262.0	14000 24000
13500	1	1	13500	248.0	13500 23500
13000	1	0	13001	231.0	13000 23000
12500	1	1	12500	221.0	12500 22500
12000	0	0	12001	150.0	12000 22000
11500	0	1	11500	141.0	11500 21500
11000	1	0	11001	199.0	11000 21000
10500	1	0	10500	192.0	10500 20500
10000	0	1	10000	126.0	10000 20000
9500	1	0	9500	182.0	9500 19500
9000	1	0	9000	161.0	9000 19000
8500	0	0	8500	103.0	8500 18500
8000	0	0	8000	78.0	8000 18000
7500	0	0	7500	53.0	7500 17500

If you look at the metadata you can see that each element has a time interval ranging from its start timestamp to a future time that lies 10000 ahead, e.g. 7500 to 17500, 8000 to 18000 and so on. The meaning of the time is, that elements that are valid at the same time should be treated together. If you look e.g. at time 10000 there are 6 events which time interval contains this value. The seventh element (with start time stamp 10500) is part of another window.

We can now combine the window with the aggregation.

Remove the running query.

Create a new PQL query (window_aggregation1) with the following input:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
windowed = TIMEWINDOW({SIZE=10000},nexmark:bid)
out = AGGREGATE({
  aggregations=[
    ['COUNT', 'price', 'COUNT_price', 'integer'],
    ['AVG', 'price', 'AVG_price'],
    ['SUM', 'price', 'SUM_price'],
    ['MIN', 'price', 'MIN_price'],
    ['MAX', 'price', 'MAX_price']
  ]
},
windowed
)
```

If you run the query the result should look like this:

COUNT_price	AVG_price	SUM_price	MIN_price	MAX_price	Metadata
20	212.4	4248.0	76.0	312.0	26000 26500
20	214.45	4289.0	76.0	312.0	25500 26000
20	220.05	4401.0	76.0	312.0	25000 25500
20	215.4	4308.0	76.0	297.0	24500 25000
20	216.05	4321.0	76.0	297.0	24000 24500
20	219.15	4383.0	76.0	297.0	23500 24000
20	217.9	4358.0	76.0	297.0	23000 23500
20	214.6	4292.0	76.0	296.0	22500 23000
20	216.7	4334.0	76.0	296.0	22000 22500
20	220.25	4405.0	76.0	296.0	21500 22000
20	215.45	4309.0	76.0	296.0	21000 21500
20	211.8	4236.0	76.0	296.0	20500 21000
20	213.5	4270.0	76.0	296.0	20000 20500
20	216.0	4320.0	126.0	296.0	19500 20000
20	210.3	4206.0	126.0	273.0	19000 19500
20	205.5	4110.0	126.0	273.0	18500 19000
20	197.0	3940.0	103.0	267.0	18000 18500
20	192.8	3856.0	78.0	267.0	17500 18000
20	183.65	3673.0	53.0	267.0	17000 17500
19	185.3684210526316	3522.0	53.0	267.0	16500 17000
18	180.83333333333334	3255.0	53.0	262.0	16000 16500
17	177.2941176470588	3014.0	53.0	262.0	15500 16000
16	173.5625	2777.0	53.0	262.0	15000 15500
15	170.53333333333333	2558.0	53.0	262.0	14500 15000
14	167.64285714285714	2347.0	53.0	262.0	14000 14500
13	160.3846153846154	2085.0	53.0	248.0	13500 14000
12	153.08333333333334	1837.0	53.0	231.0	13000 13500
11	146.0	1606.0	53.0	221.0	12500 13000
10	138.5	1385.0	53.0	199.0	12000 12500
9	137.22222222222223	1235.0	53.0	199.0	11500 12000
8	136.75	1094.0	53.0	199.0	11000 11500
7	127.85714285714286	895.0	53.0	192.0	10500 11000
6	117.16666666666667	703.0	53.0	182.0	10000 10500
5	115.4	577.0	53.0	182.0	9500 10000
4	98.75	395.0	53.0	161.0	9000 9500
3	78.0	234.0	53.0	103.0	8500 9000
2	65.5	131.0	53.0	78.0	8000 8500
1	53.0	53.0	53.0	53.0	7500 8000

You can see, that the count attribute is counted up to 20. This mean that in this case 10000 (here 10 seconds) contains 20 events. If you look at the result of the above window query, you can see that the first and the twentieth element have an overlapping time stamp, and the 21th element not. So the new window only treats element 2 to 21.

Note, in this szenario the elements are generated in a constant fashion, so the size of the window is always 20. In "real" szenarios this is typically not the case and the size of the windows in terms of containg elements differ (e.g. count the number of cars for an hour).

The window we defined is called a sliding window. The query delivers the aggregations about the bids that were posed in the last 10 seconds.

Sometimes, the result should not be in a sliding way. E.g. if a provider wants to know how many bids are posed in 10 second intervals. If the size is 10 seconds, this is called a tumbling window.

A tumbling window can be defined with one of the two parameters: ADVANCE or SLIDE. The difference is how the start time stamp of the elements in the window are treated. Let us start with ADVANCE.

Remove all running queries.

Create a new PQL script timewindow2.qry as in the following:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = TIMEWINDOW({SIZE=10000, ADVANCE=10000},nexmark:bid)
```

Running should show you something like this:

timestamp	auction	bidder	datetime	price	Metadata
23500	3	0	23501	273.0	23500 30000
23000	1	0	23000	297.0	23000 30000
22500	0	3	22501	179.0	22500 30000
22000	5	0	22001	79.0	22000 30000
21500	2	3	21501	237.0	21500 30000
21000	3	0	21001	272.0	21000 30000
20500	4	2	20500	158.0	20500 30000
20000	5	3	20001	76.0	20000 30000
19500	1	0	19500	296.0	19500 20000
19000	3	1	19001	257.0	19000 20000
18500	1	2	18501	273.0	18500 20000
18000	0	1	18001	162.0	18000 20000
17500	2	1	17500	236.0	17500 20000
17000	0	1	17001	151.0	17000 20000
16500	1	0	16500	267.0	16500 20000
16000	3	0	16000	241.0	16000 20000
15500	3	1	15500	237.0	15500 20000
15000	3	2	15000	219.0	15000 20000
14500	2	1	14500	211.0	14500 20000
14000	1	1	14000	262.0	14000 20000
13500	1	1	13501	248.0	13500 20000
13000	1	0	13001	231.0	13000 20000
12500	1	1	12501	221.0	12500 20000
12000	0	0	12001	150.0	12000 20000
11500	0	1	11500	141.0	11500 20000
11000	1	0	11001	199.0	11000 20000
10500	1	0	10501	192.0	10500 20000
10000	0	1	10001	126.0	10000 20000
9500	1	0	9500	182.0	9500 10000
9000	1	0	9000	161.0	9000 10000
8500	0	0	8501	103.0	8500 10000
8000	0	0	8001	78.0	8000 10000
7500	0	0	7500	53.0	7500 10000

You can see, that the timestamps are different. Each events belonging to the same window has the same end time stamp (e.g. 10.000, 20.000, 30.000).

If you run the following aggregation query (window_aggregation2)

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
windowed = TIMEWINDOW({SIZE=10000, ADVANCE=10000},nexmark:bid)
out = AGGREGATE({
  aggregations=[
    ['COUNT', 'price', 'COUNT_price', 'integer'],
    ['AVG', 'price', 'AVG_price'],
    ['SUM', 'price', 'SUM_price'],
    ['MIN', 'price', 'MIN_price'],
    ['MAX', 'price', 'MAX_price']
  ],
  windowed
})
```

you will see the following output (when no other query is running!):

COUNT_price	AVG_price	SUM_price	MIN_price	MAX_price	Metadata
8	196.375	1571.0	76.0	297.0	23500 30000
7	185.42857142857142	1298.0	76.0	297.0	23000 23500
6	166.83333333333334	1001.0	76.0	272.0	22500 23000
5	164.4	822.0	76.0	272.0	22000 22500
4	185.75	743.0	76.0	272.0	21500 22000
3	168.66666666666666	506.0	76.0	272.0	21000 21500
2	117.0	234.0	76.0	158.0	20500 21000
1	76.0	76.0	76.0	76.0	20000 20500
20	216.0	4320.0	126.0	296.0	19500 20000
19	211.78947368421052	4024.0	126.0	273.0	19000 19500
18	209.27777777777777	3767.0	126.0	273.0	18500 19000
17	205.52941176470588	3494.0	126.0	267.0	18000 18500
16	208.25	3332.0	126.0	267.0	17500 18000
15	206.4	3096.0	126.0	267.0	17000 17500
14	210.35714285714286	2945.0	126.0	267.0	16500 17000
13	206.0	2678.0	126.0	262.0	16000 16500
12	203.08333333333334	2437.0	126.0	262.0	15500 16000
11	200.0	2200.0	126.0	262.0	15000 15500
10	198.1	1981.0	126.0	262.0	14500 15000
9	196.66666666666666	1770.0	126.0	262.0	14000 14500
8	188.5	1508.0	126.0	248.0	13500 14000
7	180.0	1260.0	126.0	231.0	13000 13500
6	171.5	1029.0	126.0	221.0	12500 13000
5	161.6	808.0	126.0	199.0	12000 12500
4	164.5	658.0	126.0	199.0	11500 12000
3	172.33333333333334	517.0	126.0	199.0	11000 11500
2	159.0	318.0	126.0	192.0	10500 11000
1	126.0	126.0	126.0	126.0	10000 10500
5	115.4	577.0	53.0	182.0	9500 10000
4	98.75	395.0	53.0	161.0	9000 9500
3	78.0	234.0	53.0	103.0	8500 9000
2	65.5	131.0	53.0	78.0	8000 8500
1	53.0	53.0	53.0	53.0	7500 8000

As you can see, the aggregations count up to 20 and then start again at 1. This is not exactly, what the provider wanted. The problem here is, that the elements have different start time stamps and only those parts of the elements are aggregated where the time intervals overlap.

A way to cope with this is to use another parameter called SLIDE. With slide all start timestamp are set to the same value. This can be interpreted as reducing the granularity of the time domain.

So remove all queries. A start a new PQL query (timewindow3)

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = TIMEWINDOW({SIZE=10000, SLIDE=10000},nexmark:bid)
```

After starting this query, you should see something like:

timestamp	auction	bidder	datetime	price	Metadata
23500	3	0	23501	273.0	20000 30000
23000	1	0	23001	297.0	20000 30000
22500	0	3	22500	179.0	20000 30000
22000	5	0	22001	79.0	20000 30000
21500	2	3	21500	237.0	20000 30000
21000	3	0	21000	272.0	20000 30000
20500	4	2	20500	158.0	20000 30000
20000	5	3	20000	76.0	20000 30000
19500	1	0	19500	296.0	10000 20000
19000	3	1	19000	257.0	10000 20000
18500	1	2	18500	273.0	10000 20000
18000	0	1	18000	162.0	10000 20000
17500	2	1	17500	236.0	10000 20000
17000	0	1	17000	151.0	10000 20000
16500	1	0	16501	267.0	10000 20000
16000	3	0	16001	241.0	10000 20000
15500	3	1	15500	237.0	10000 20000
15000	3	2	15001	219.0	10000 20000
14500	2	1	14501	211.0	10000 20000
14000	1	1	14001	262.0	10000 20000
13500	1	1	13500	248.0	10000 20000
13000	1	0	13000	231.0	10000 20000
12500	1	1	12501	221.0	10000 20000
12000	0	0	12001	150.0	10000 20000
11500	0	1	11500	141.0	10000 20000
11000	1	0	11000	199.0	10000 20000
10500	1	0	10501	192.0	10000 20000
10000	0	1	10001	126.0	10000 20000
9500	1	0	9501	182.0	0 10000
9000	1	0	9001	161.0	0 10000
8500	0	0	8500	103.0	0 10000
8000	0	0	8000	78.0	0 10000
7500	0	0	7501	53.0	0 10000

Now you can see, that for one window all time intervals are the same.

If we now run the following query (window_aggregation3):

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
windowed = TIMEWINDOW({SIZE=10000, SLIDE=10000},nexmark:bid)
out = AGGREGATE({
  aggregations=[
    ['COUNT', 'price', 'COUNT_price', 'integer'],
    ['AVG', 'price', 'AVG_price'],
    ['SUM', 'price', 'SUM_price'],
    ['MIN', 'price', 'MIN_price'],
    ['MAX', 'price', 'MAX_price']
  ]
},
windowed
)
```

You will get the following output:

COUNT_price	AVG_price	SUM_price	MIN_price	MAX_price	Metadata
6	155.16666666666666	931.0	43.0	272.0	70000 80000
20	152.45	3049.0	34.0	395.0	60000 70000
20	174.15	3483.0	16.0	393.0	50000 60000
20	203.75	4075.0	32.0	373.0	40000 50000
20	174.8	3496.0	4.0	345.0	30000 40000
20	200.75	4015.0	76.0	322.0	20000 30000
20	216.0	4320.0	126.0	296.0	10000 20000
5	115.4	577.0	53.0	182.0	0 10000

Note: You could ignore the first line, it was created because when the query was stopped.

Here you can see the aggregation over 10 second intervals.

Element Based Window

Another way to define a window, is based on elements, e.g. the last 10 elements received. In Odysseus, the element based window is implemented with timestamps, too.

Create a new PQL script:

```
#PARSER PQL
#TRANSCFG Standard
#ADDQUERY
out = ELEMENTWINDOW({SIZE=10}, nexmark:bid)
```

Here you can see the operator ELEMENTWINDOW with the parameter SIZE, that defines a window containing 10 elements.

If you start the query, you will see:

timestamp	auction	bidder	datetime	price	Metadata
21000	3	0	21000	272.0	21000 26000
20500	4	2	20500	158.0	20500 25500
20000	5	3	20000	76.0	20000 25000
19500	1	0	19501	296.0	19500 24500
19000	3	1	19001	257.0	19000 24000
18500	1	2	18508	273.0	18500 23500
18000	0	1	18001	162.0	18000 23000
17500	2	1	17501	236.0	17500 22500
17000	0	1	17000	151.0	17000 22000
16500	1	0	16501	267.0	16500 21500
16000	3	0	16000	241.0	16000 21000
15500	3	1	15500	237.0	15500 20500
15000	3	2	15000	219.0	15000 20000
14500	2	1	14502	211.0	14500 19500
14000	1	1	14000	262.0	14000 19000
13500	1	1	13501	248.0	13500 18500
13000	1	0	13000	231.0	13000 18000
12500	1	1	12501	221.0	12500 17500
12000	0	0	12000	150.0	12000 17000
11500	0	1	11500	141.0	11500 16500
11000	1	0	11001	199.0	11000 16000
10500	1	0	10501	192.0	10500 15500
10000	0	1	10001	126.0	10000 15000
9500	1	0	9501	182.0	9500 14500
9000	1	0	9000	161.0	9000 14000
8500	0	0	8500	103.0	8500 13500
8000	0	0	8000	78.0	8000 13000
7500	0	0	7500	53.0	7500 12500

If you look at the time stamps, you will see that the end timestamp of the first element is 12500. This is exactly the start time stamp of the eleventh element, that should not be in the same window as the first element (size = 10). To be able to determine the end time stamp the output of the first element, the system needs to be wait until the eleventh element has arrived at the system. So, using element based windows may lead to a higher latency.

In the same way, ADVANCE and SLIDE are used in the time based windows, these parameters can be used here.

A special case in element based windows is the partition parameter. Typically, the size is defined over all elements in the stream. With the partition parameter you can define an attribute that partitions the stream. In this case, the window will be defined over elements where the value of this attribute are the same.

Note: If the stream has a low data rate, it may take very long before the first elements are created!

TODO: Example for Partition Window

Group By

TODO and Partition Window

Predicate Based Window

The most generic way to define windows