

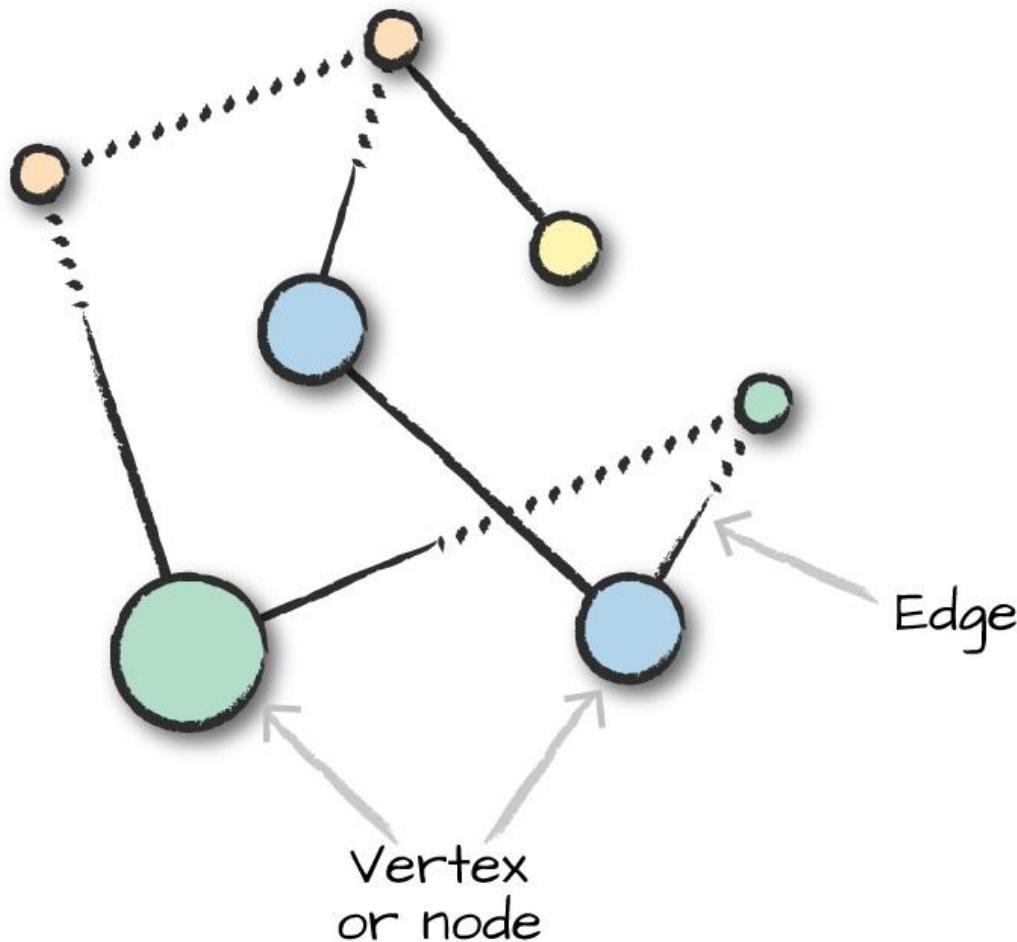
Graph Analytics in Spark

Graph analytics: Introduction

Graph analytics

- Graphs are data structures composed of nodes and edges
 - Nodes/vertexes are denoted as $V=\{v_1, v_2, \dots, v_n\}$ and edges are denoted as $E=\{e_1, e_2, \dots, e_n\}$
 - Graph analytics is the process of analyzing relationships between vertexes and edges

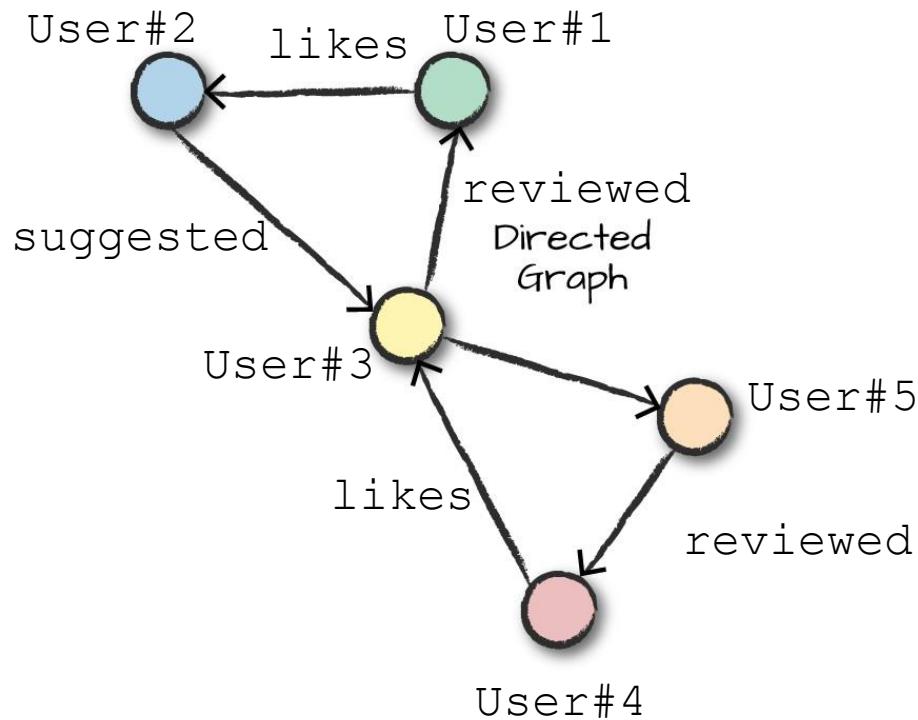
Graph analytics



Vertexes, edges and weights

- Graphs are undirected if edges do not have a direction
- Otherwise they are called directed graphs
- Vertexes and edges can have data associated with them
 - weight/label
 - e.g., an edge weight may represent the strength of the relationship
 - e.g., a vertex label may be the string associated with the name of the vertex

Vertices, edges and weights

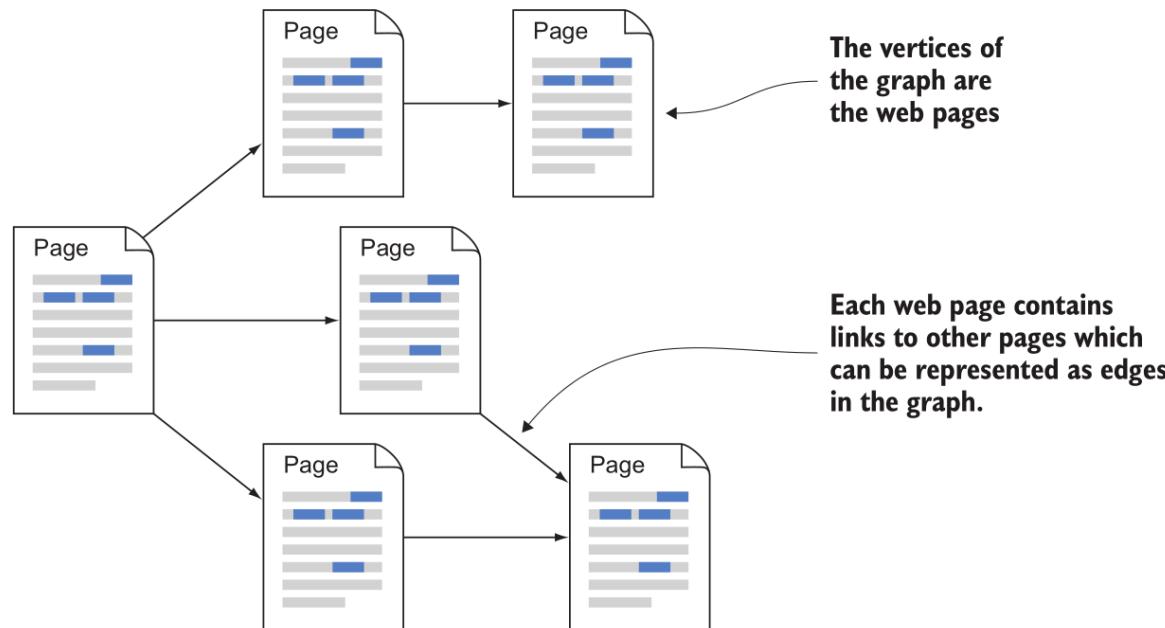


Why graph analytics?

- Graphs are natural way of describing relationships
- Practical example of analytics over graphs
 - Ranking web pages (Google PageRank)
 - Detecting group of friends
 - Determine importance of infrastructure in electrical networks
 - ...

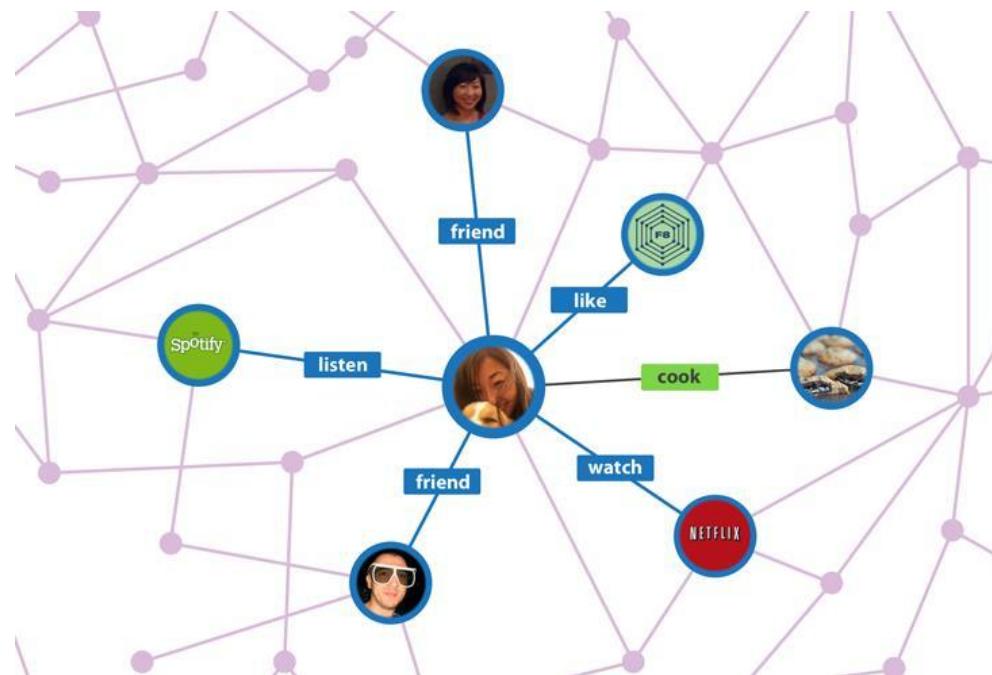
Graph structure in the web

■ Importance and rank of web pages



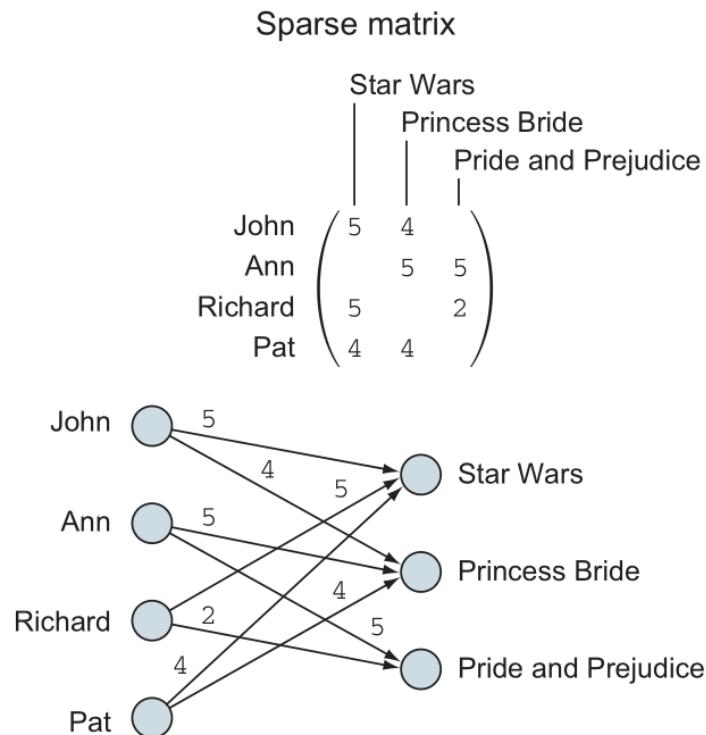
Graph structure in the web

- Social network structure and web usage

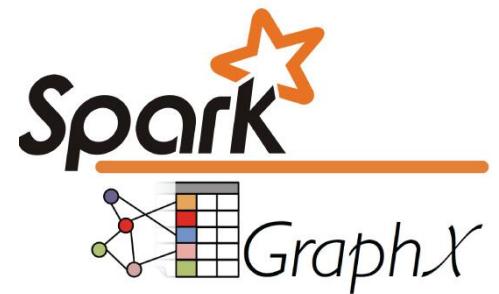


Graph structure in the web

■ Movies watched by users

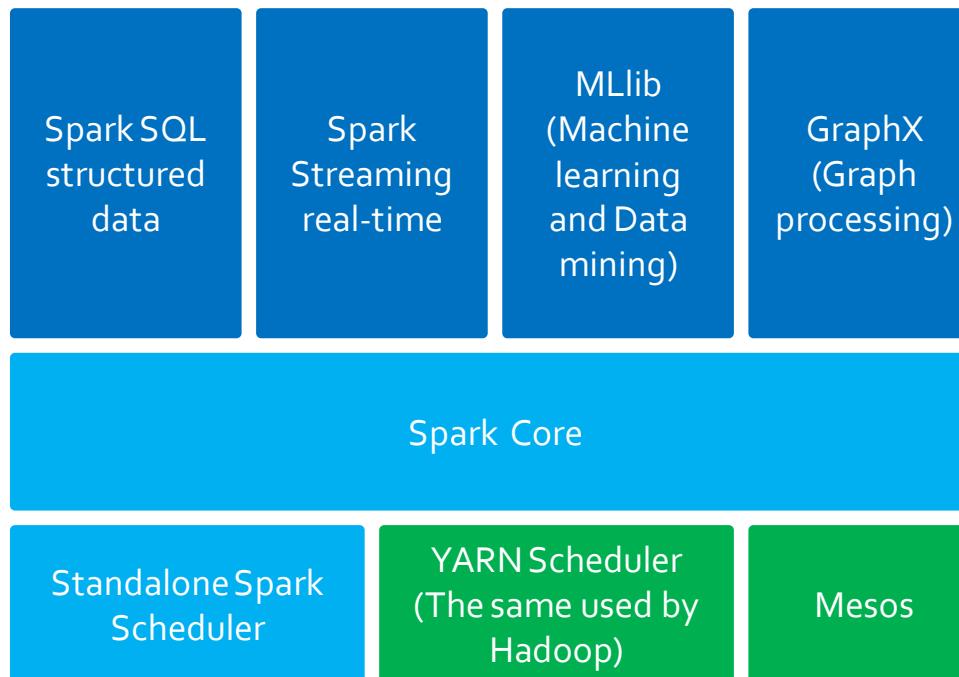


Spark GraphX and GraphFrames



GraphX

- Spark RDD-based library for performing graph processing
- Core part of Spark



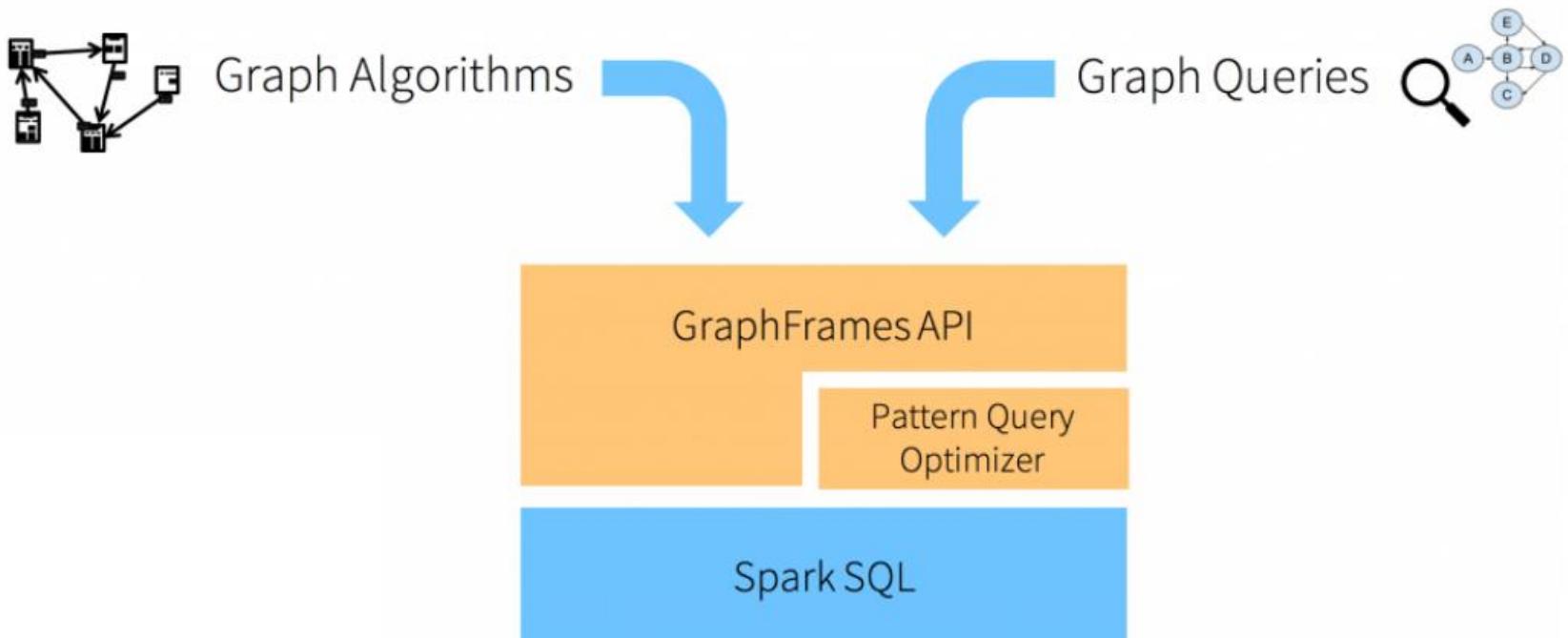
GraphX

- Low level interface with RDD
- Very powerful
 - Many application and libraries built on top of it
- However, not easy to use or optimize
- No Python version of the APIs

GraphFrames

- Library DataFrame-based for performing graph processing
- Spark external package built on top of GraphX
 - https://graphframes.github.io/graphframes/docs/_site/index.html

GraphFrames



Building and querying graphs with GraphFrames

Building a graph

- Define vertexes and edges of the graph
 - Vertexes and edges are represented by means of records inside DataFrames with specifically named columns
 - One DataFrame for the definition of the vertexes of the graph
 - One DataFrame for the definition of the edges of the graph

Building a graph

- The DataFrames that are used to represent
nodes/vertexes
 - Contain **one record per vertex**
 - Must contain a column named "**id**" that stores unique vertex IDs
 - Can contain other columns that are used to characterize vertexes

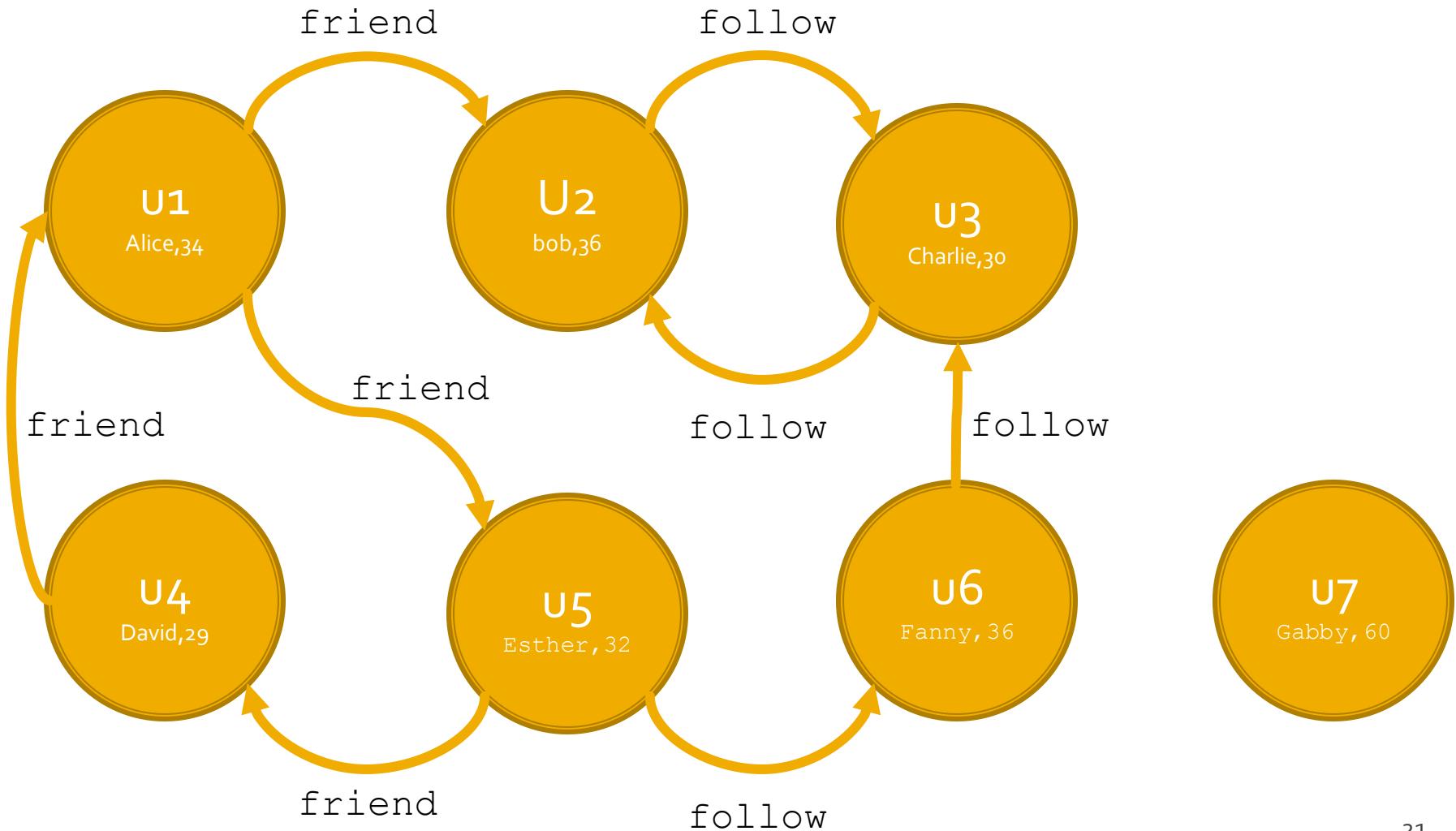
Building a graph

- The DataFrames that are used to represent edges
 - Contain **one record per edge**
 - Must contain two columns "**src**" and "**dst**" storing source vertex IDs and destination vertex IDs of edges
 - Can contain other columns that are used to characterize edges

Building a graph

- Create a graph of type
graphframes.graphframe.GraphFrame by
invoking the constructor **GraphFrame(v,e)**
 - v
 - The DataFrame containing the definition of the vertexes
 - e
 - The DataFrame containing the definition of the edges
- Graphs in graphframes are **directed graphs**

Building a graph: Example



Building a graph: Example

Vertex DataFrame

id	name	age
u1	Alice	34
u2	Bob	36
u3	Charlie	30
u4	David	29
u5	Esther	32
u6	Fanny	36
u7	Gabby	60

Edge DataFrame

src	dst	relationship
u1	u2	friend
u2	u3	follow
u3	u2	follow
u6	u3	follow
u5	u6	follow
u5	u4	friend
u4	u1	friend
u1	u5	friend

Building a graph: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Building a graph: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Directed vs undirected edges

- In undirected graphs the edges indicate a two-way relationship (each edge can be traversed in both directions)
- In GraphX you could use `to_undirected()` to create an undirected copy of the Graph
- Unfortunately **GraphFrames does not support it**
 - You can convert your graph by applying a `flatMap` function over the edges of the directed graph that creates symmetric edges and then create a new `GraphFrame`

Cache graphs

- As with RDD and DataFrame, you can **cache** graphs in GraphFrame
 - Convenient if the same (complex) graph result of (multiple) transformations is used multiple times in the same application
 - Simply invoke **cache()** on the GraphFrame you want to cache
 - It **persists** the DataFrame-based representation of vertexes and edges of the graph

Querying the graph

- Some specific methods are provided to execute queries on graphs
 - `filterVertices(condition)`
 - `filterEdges(condition)`
 - `dropIsolatedVertices()`
- The returned result is the filtered version of the input graph

Querying the graph: filterVertices

- **filterVertices(condition)**
 - condition contains an SQL-like condition on the values of the attributes of the vertexes
 - E.g., “age>35”
 - Selects only the vertexes for which the specified condition is satisfied and returns a new graph with only the subset of selected vertexes

Querying the graph: filterEdges

- **filterEdges(condition)**

- condition contains an SQL-like condition on the values of the attributes of the edges
 - E.g., "relationship='friend'"
- Selects only the edges for which the specified condition is satisfied and returns a new graph with only the subset of selected edges

Querying the graph: dropIsolatedVertices

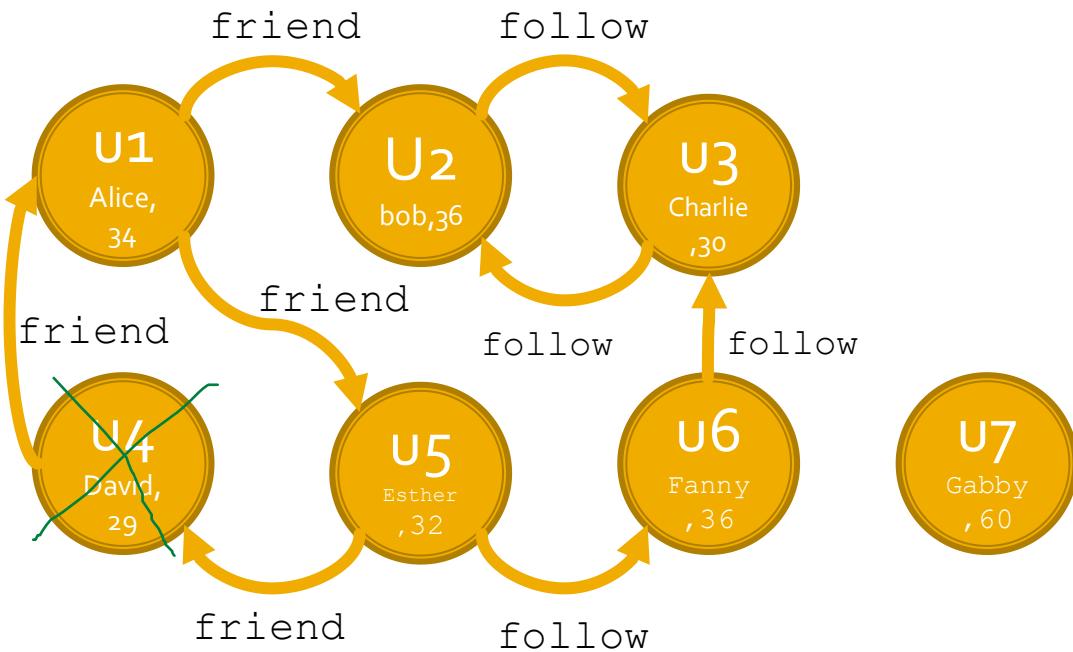
- **dropIsolatedVertices()**
 - Drops the vertexes that are not connected with any other node and returns a new graph without the dropped nodes

Querying the graph: Example 1

- Given the input graph, create a new subgraph including
 - Only the vertexes associated with users characterized by age between 30 and 50
 - Only the edges representing the friend relationship
 - Drop isolated vertexes

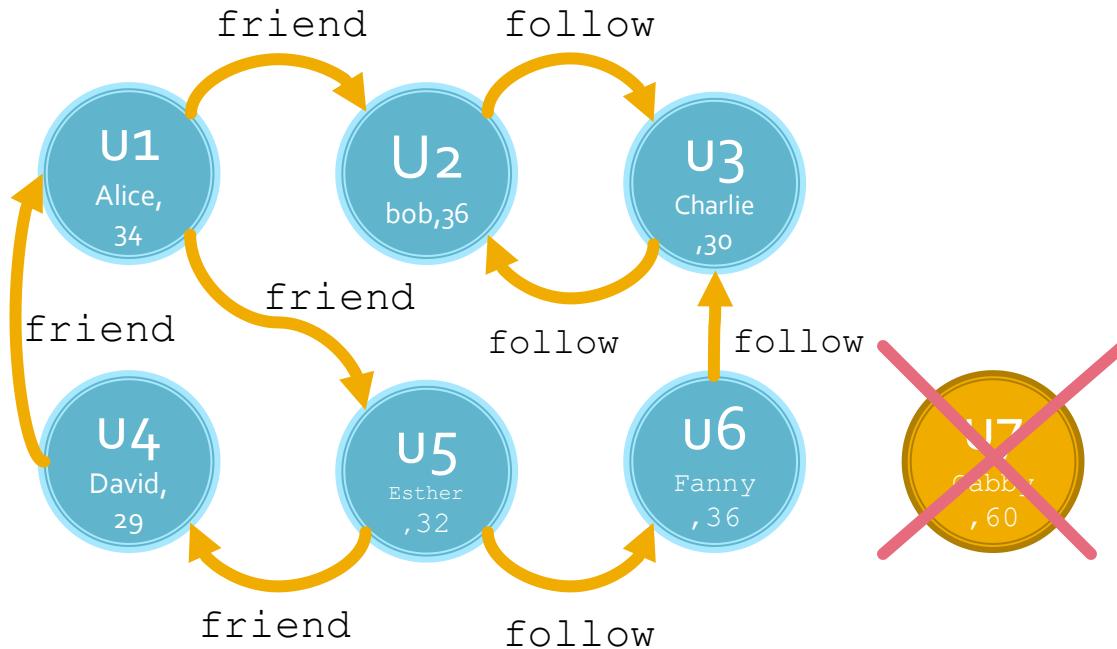
Querying the graph: Example 1

Input graph



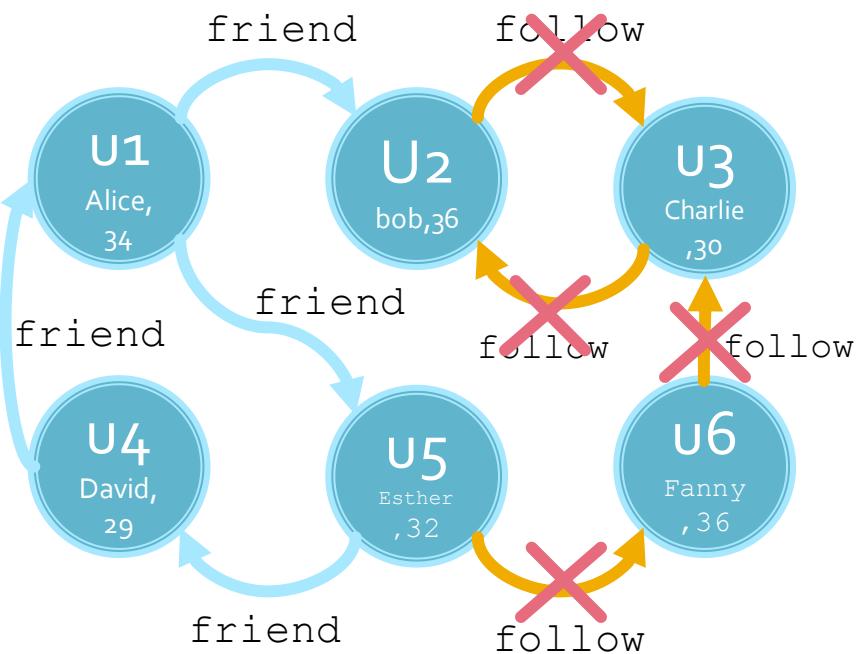
Querying the graph: Example 1

Filter vertexes



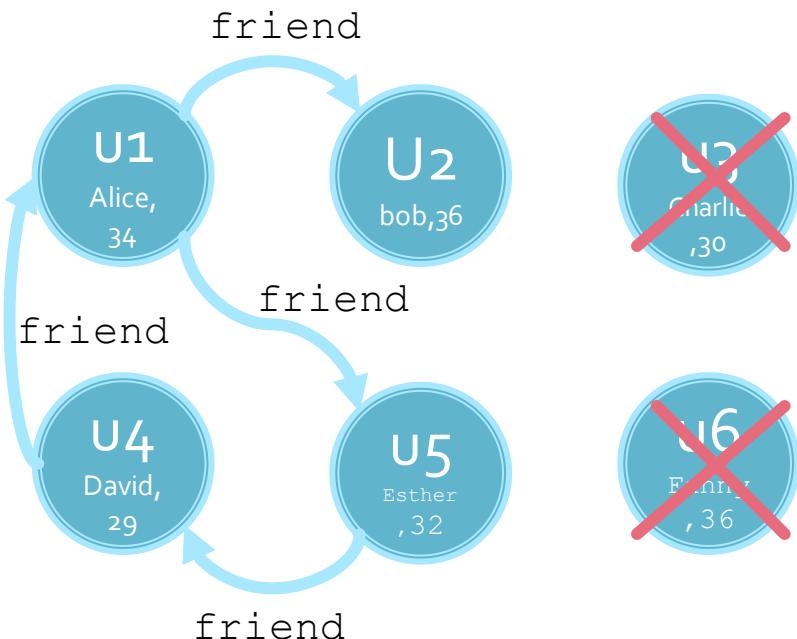
Querying the graph: Example 1

Filter edges



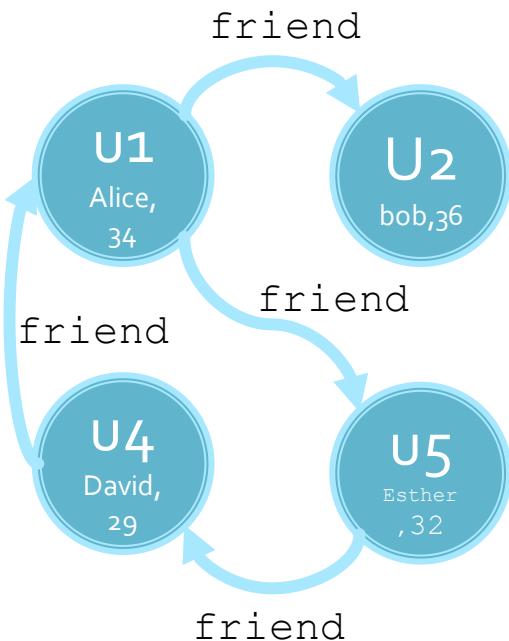
Querying the graph: Example 1

Drop isolated vertexes



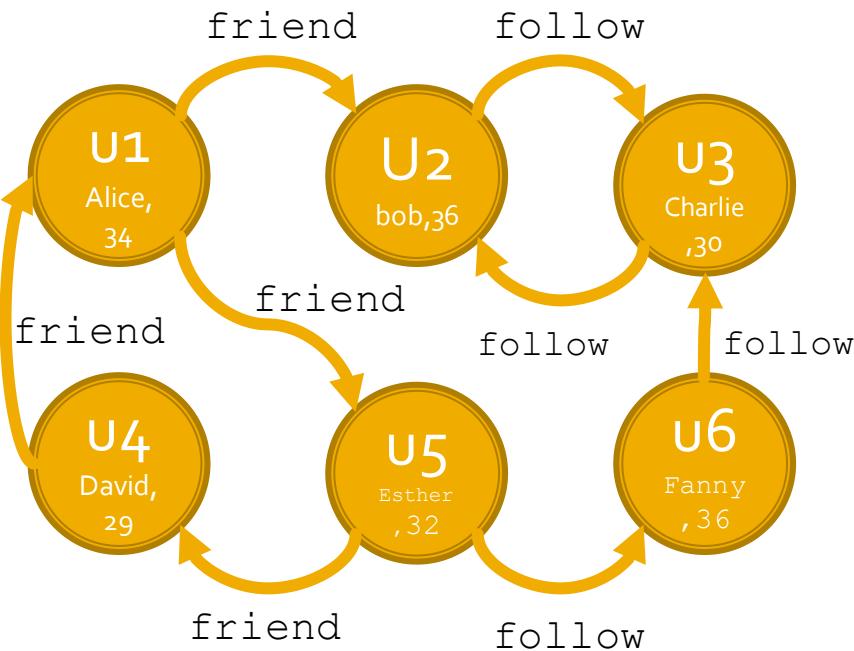
Querying the graph: Example 1

Output graph

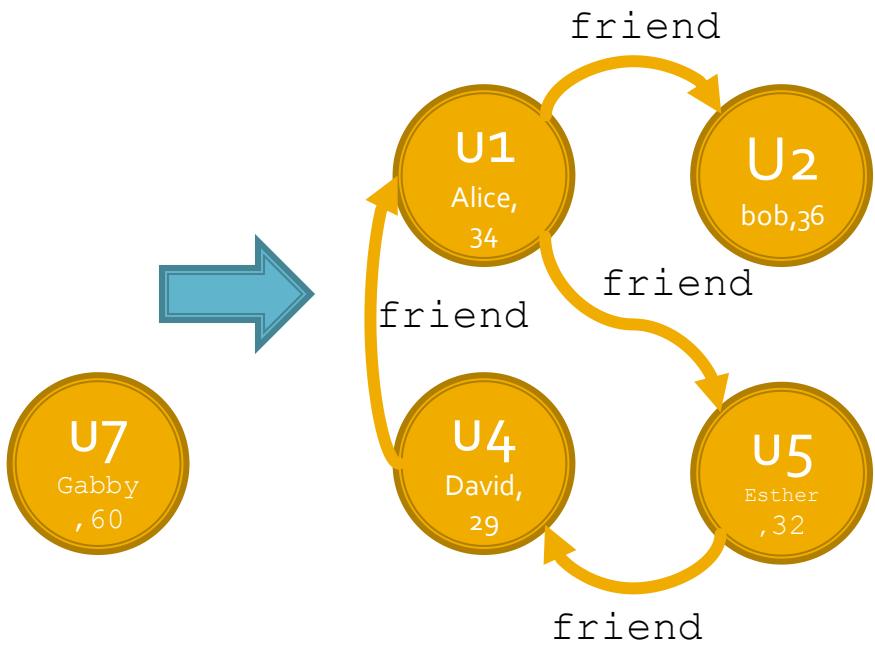


Querying the graph: Example 1

Input graph



Output graph



Querying the graph: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Querying the graph: Example 1

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Querying the graph: Example 1

```
selectedUsersandFriendRelGraph = g\  
.filterVertices("age>=30 AND age<=50")\  
.filterEdges("relationship='friend'")  
.dropIsolatedVertices()      it is like a query
```

Querying the graph

- Given a GraphFrame, we can easily access its vertexes and edges
 - `g.vertices` returns the DataFrame associated with the vertexes of the input graph
 - `g.edges` returns the DataFrame associated with the edges of the input graph

Querying the graph

- All the standard DataFrame transformations/actions are available also for the DataFrames that are used to store vertexes and edges
 - For example, the number of vertexes and the number of edges can be computed by invoking the count() action on the DataFrames vertices and edges, respectively

Querying the graph: Example 2

- Given the input graph
 - Count how many vertexes and edges has the graph
 - Find the smallest value of age (i.e., the age of the youngest user in the graph)
 - Count the number of edges of type "follow" in the graph

Querying the graph: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Querying the graph: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Querying the graph: Example 2

```
# Count how many vertexes and edges has the graph
print("Number of vertexes: ",g.vertices.count())
print("Number of edges: ",g.edges.count())

# Print on the standard output the smallest value of age
# (i.e., the age of the youngest user in the graph)
g.vertices.agg({"age":"min"}).show()

# Print on the standard output
# the number of "follow" edges in the graph.
numFollows = g.edges.filter("relationship = 'follow' ").count()

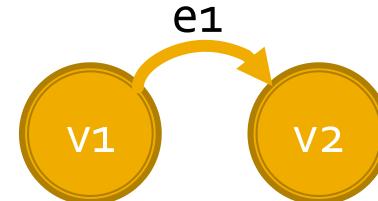
print(numFollows)
```

Motif finding

- Motif finding refers to searching for structural patterns in graphs
- A simple Domain-Specific Language (DSL) is used to specify the structure of the patterns we are interested in
 - The paths/subgraphs in the graph matching the specified structural pattern are selected

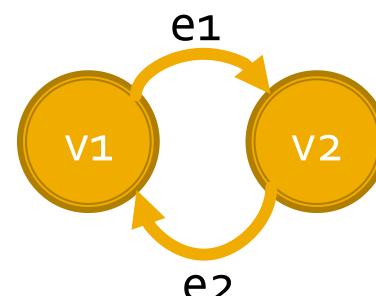
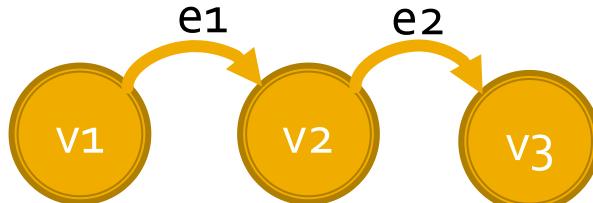
DSL for Motif finding

- The **basic unit** of a pattern is a connection between vertexes
 - $(v_1) - [e_1] \rightarrow (v_2)$ means
 - An arbitrary edge $[e_1]$ from an arbitrary vertex (v_1) to another arbitrary vertex (v_2)
- **Edges** are denoted by **square brackets**
 - $[e_1]$
- **Vertexes** are expressed by **round brackets**
 - $(v_1), (v_2)$



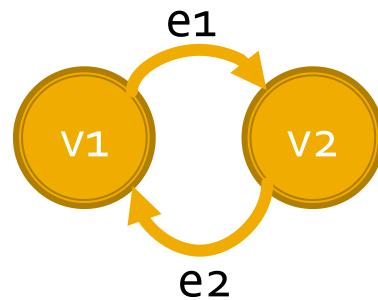
DSL for Motif finding

- Patterns are chains of basic units
 - $(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_3)$
means
 - An arbitrary edge from an arbitrary vertex v_1 to another arbitrary vertex v_2 and another arbitrary edge from v_2 to another arbitrary vertex v_3
 - v_3 and v_1 can be the same vertex



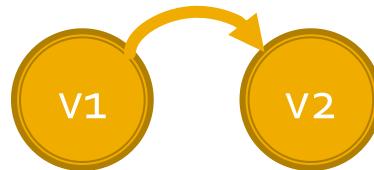
DSL for Motif finding

- The same vertex name is used in a pattern to have a reference to the same vertex
 - $(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$ means
 - An arbitrary edge from an arbitrary vertex v_1 to another arbitrary vertex v_2 and vice-versa



DSL for Motif finding

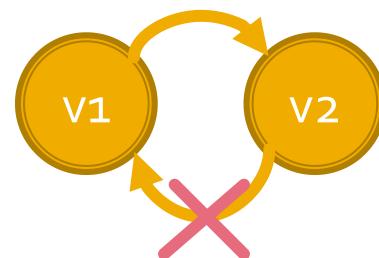
- It is acceptable to omit names for vertices or edges in patterns when not needed
 - $(v_1)-[]\rightarrow(v_2)$ expresses an arbitrary edge between two arbitrary vertexes v_1, v_2 but does not assign a name to the edge



- These are called **anonymous** vertexes and edges

DSL for Motif finding

- A basic unit (an edge between two vertexes) can be negated to indicate that the edge should not be present in the graph
 - $(v1)-[]\rightarrow(v2); !(v2)-[]\rightarrow(v1)$
means
 - Edges from $v1$ to $v2$ but no edges from $v2$ to $v1$



DSL for Motif finding

- The **find(motif)** method of GraphFrame is used to select motifs
 - **motif**
 - DSL representation of the structural pattern

DSL for Motif finding

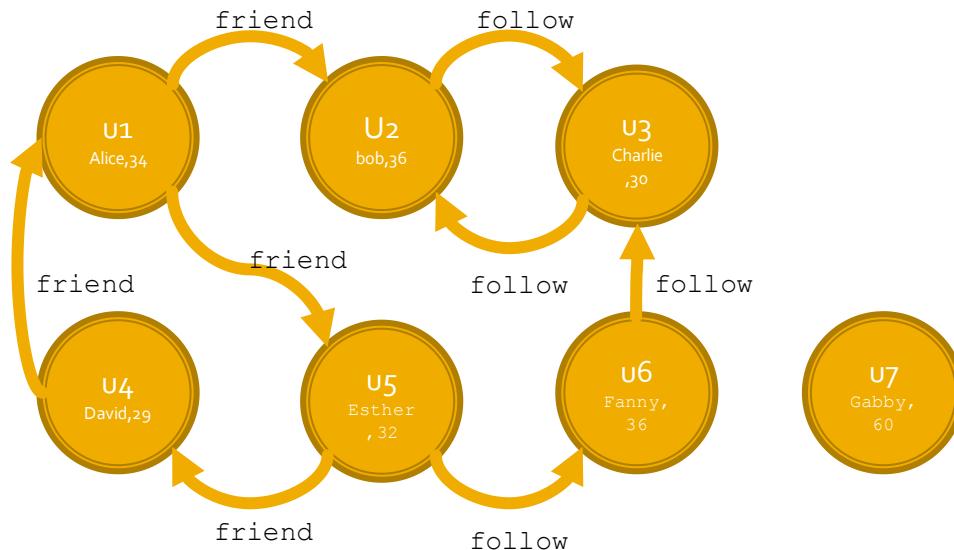
- **find()** returns a **DataFrame** of all the **paths** matching the structural motif/pattern
 - One path per record
 - The returned DataFrame will have a column for each of the named elements (vertexes and edges) in the structural pattern/motif
 - Each column is a struct
 - The fields of each struct are the labels/features of the associated vertex or edge
 - It can return duplicate rows/records
 - If there are many paths connecting the same nodes

DSL for Motif finding

- More complex queries on the structure and content of the patterns can be expressed by applying filters to the result DataFrame
 - i.e., more complex queries can be applied by combining find() and filter()

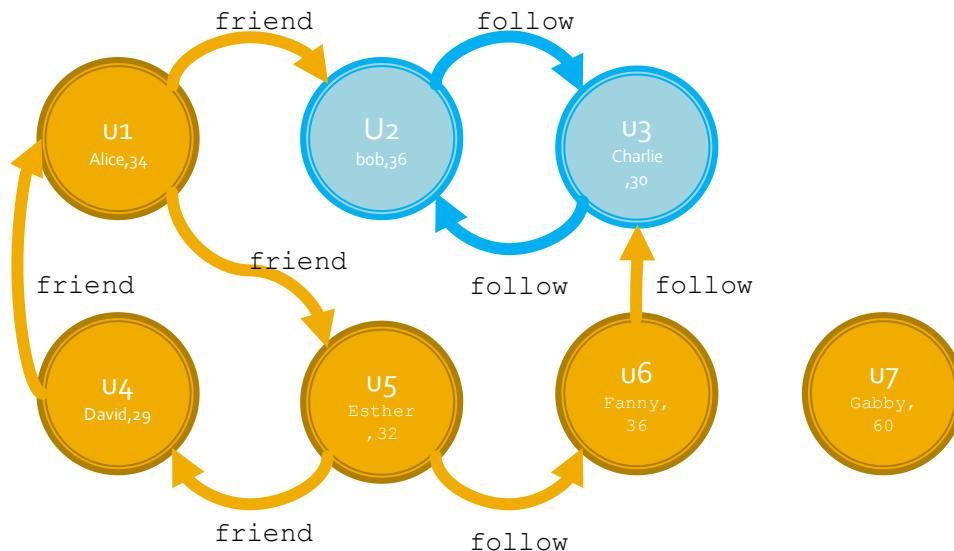
Motif finding: Example 1

- Find the paths/subgraphs matching the pattern
 $(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$
- Store the result in a DataFrame



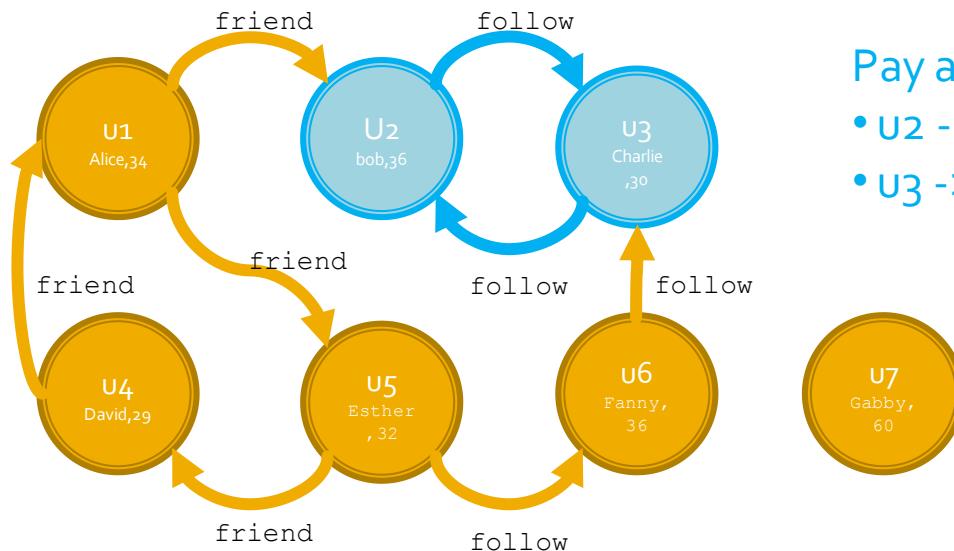
Motif finding: Example 1

- Find the paths/subgraphs matching the pattern
 $(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$
- Store the result in a DataFrame



Motif finding: Example 1

- Find the paths/subgraphs matching the pattern
 $(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$
- Store the result in a DataFrame



Pay attention that two paths are returned:

- U2 → follow → U3 → follow → U2
- U3 → follow → U2 → follow → U3

Motif finding: Example 1

- Find the paths/subgraphs matching the pattern

$$(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$$

- Content of the returned DataFrame

v1	e1	v2	e2
[u2, Bob, 36]	[u2, u3, follow]	[u3, Charlie, 30]	[u3, u2, follow]
[u3, Charlie, 30]	[u3, u2, follow]	[u2, Bob, 36]	[u2, u3, follow]

Motif finding: Example 1

- Find the paths/subgraphs matching the pattern

$(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$

- Content of the returned DataFrame

v1	e1	v2	e2
[u2, Bob, 36]	[u2, u3, follow]	[u3, Charlie, 30]	[u3, u2, follow]
[u3, Charlie, 30]	[u3, u2, follow]	[u2, Bob, 36]	[u2, u3, follow]

There is one column for each (distinct) named vertex and edge of the structural pattern

Motif finding: Example 1

- Find the paths/subgraphs matching the pattern

$$(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$$

- Content of the returned DataFrame

v1	e1	v2	e2
[u2, Bob, 36]	[u2, u3, follow]	[u3, Charlie, 30]	[u3, u2, follow]
[u3, Charlie, 30]	[u3, u2, follow]	[u2, Bob, 36]	[u2, u3, follow]

The records are associated with the vertexes and edges of the selected paths

Motif finding: Example 1

- Find the paths/subgraphs matching the pattern

$$(v_1) - [e_1] \rightarrow (v_2); (v_2) - [e_2] \rightarrow (v_1)$$

- Content of the returned DataFrame

v1	e1	v2	e2
[u2, Bob, 36]	[u2, u3, follow]	[u3, Charlie, 30]	[u3, u2, follow]
[u3, Charlie, 30]	[u3, u2, follow]	[u2, Bob, 36]	[u2, u3, follow]

All columns are associated with the data type “struct”.

Each struct has the same “schema/features” of the associated vertex or edge.

Motif finding: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Motif finding: Example 1

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

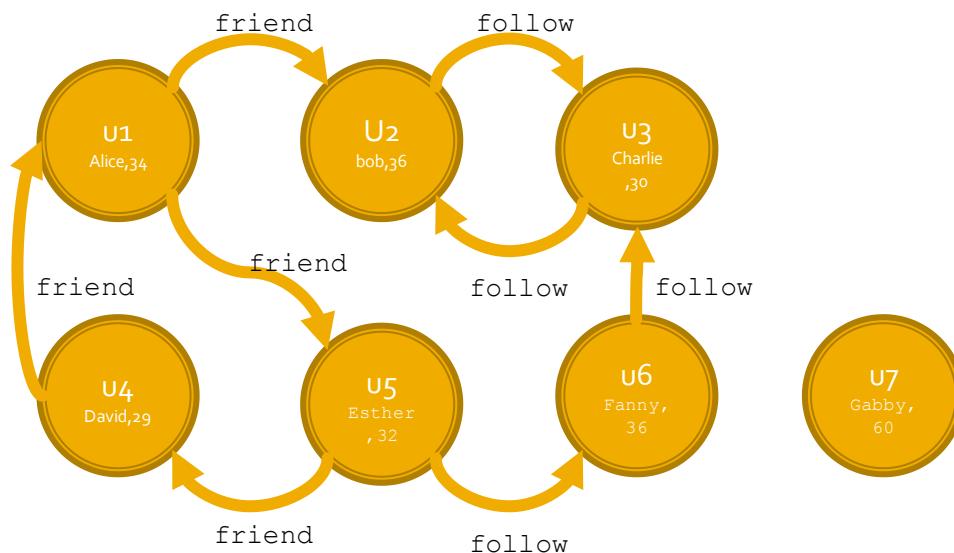
# Create the graph
g = GraphFrame(v, e)
```

Motif finding: Example 1

```
# Retrieve the motifs associated with the pattern  
# vertex -> edge -> vertex -> edge ->vertex  
motifs = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v1)")
```

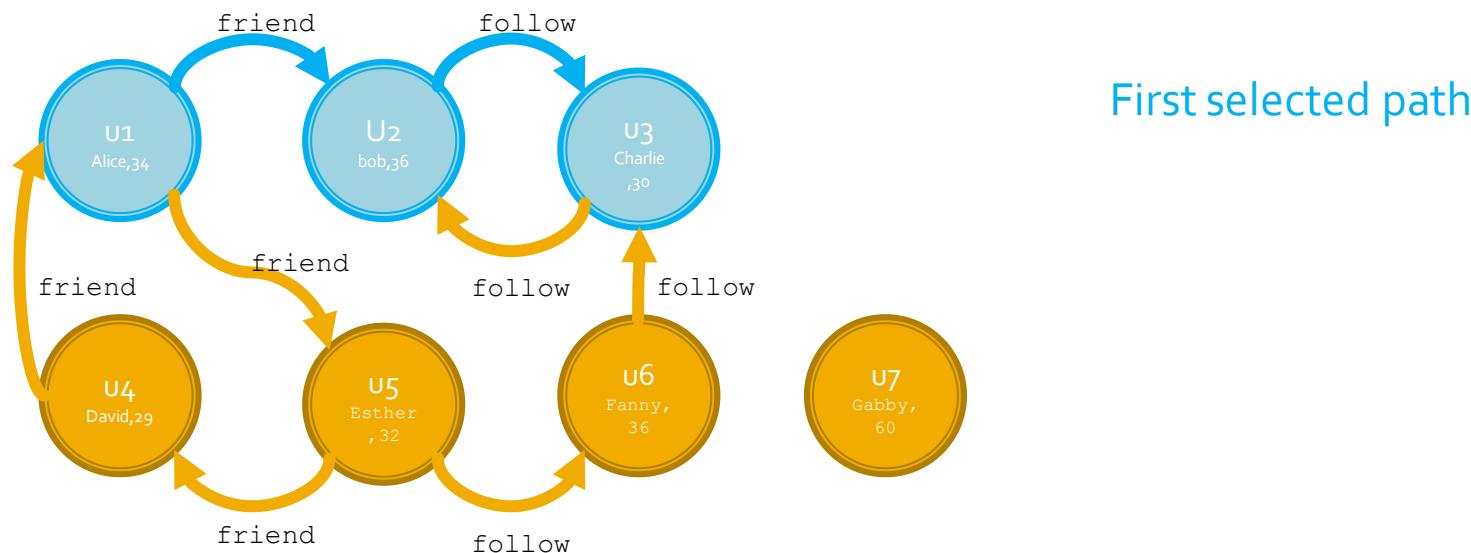
Motif finding: Example 2

- Find the paths/subgraphs matching the pattern
 $(v_1)\text{-[friend]} \rightarrow (v_2); (v_2)\text{-[follow]} \rightarrow (v_3)$
- Store the result in a DataFrame



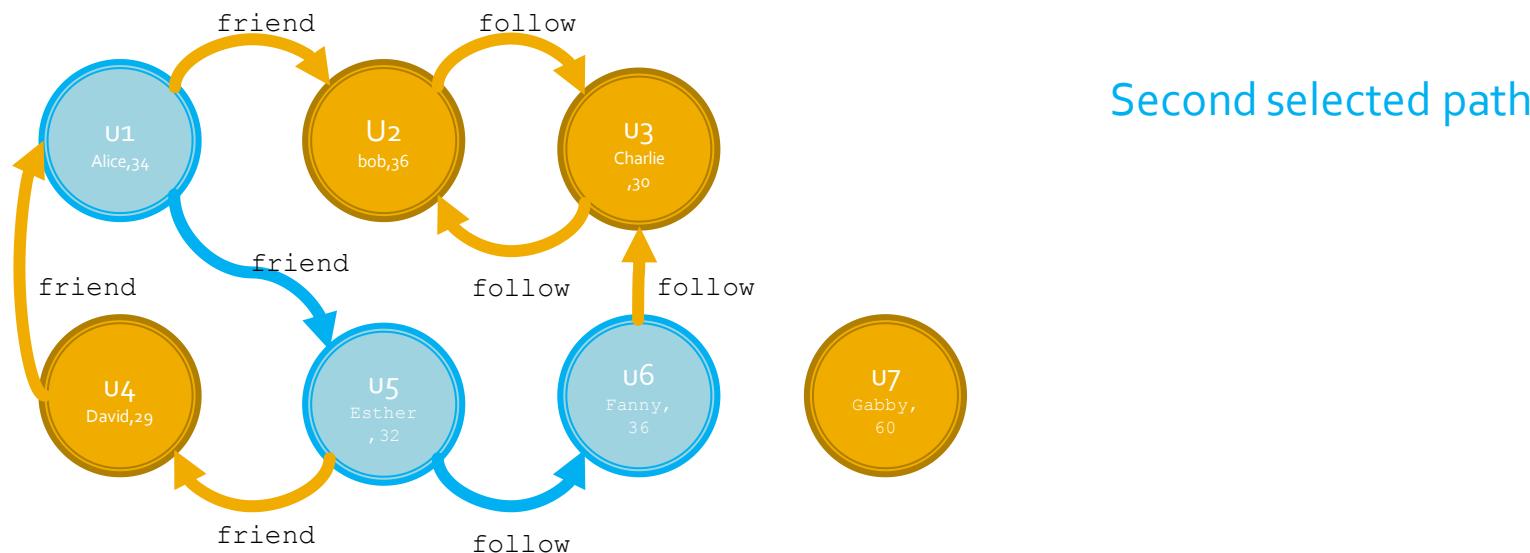
Motif finding: Example 2

- Find the paths/subgraphs matching the pattern
 $(v_1)\text{- [friend]} \rightarrow (v_2); (v_2)\text{- [follow]} \rightarrow (v_3)$
- Store the result in a DataFrame



Motif finding: Example 2

- Find the paths/subgraphs matching the pattern
 $(v_1)\text{- [friend]} \rightarrow (v_2); (v_2)\text{- [follow]} \rightarrow (v_3)$
- Store the result in a DataFrame



Motif finding: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Motif finding: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Motif finding: Example 2

```
# Retrieve the motifs associated with the pattern
# vertex -> edge -> vertex -> edge ->vertex
motifs = g.find("(v1)-[friend]->(v2); (v2)-[follow]->(v3)")

# Filter the motifs (the content of the motifs DataFrame)
# Select only the ones matching the pattern
# vertex -> friend-> vertex -> follow ->vertex
motifsFriendFollow = motifs\
.filter("friend.relationship='friend' AND follow.relationship='follow' ")
```

Motif finding: Example 2

```
# Retrieve the motifs associated with the pattern  
# vertex -> edge -> vertex -> edge ->vertex  
motifs = g.find("(v1)-[friend]->(v2); (v2)-[follow]->(v3)")  
  
# Filter the motifs (the content of the motifs DataFrame)  
# Select only the ones matching the pattern  
# vertex -> friend-> vertex -> follow ->vertex  
motifsFriendFollow = motifs\  
.filter("friend.relationship='friend' AND follow.relationship='follow' ")
```

Columns friend and follow are structs with three fields/attributes

- src
- dst
- relationship

Motif finding: Example 2

```
# Retrieve the motifs associated with the pattern  
# vertex -> edge -> vertex -> edge ->vertex  
motifs = g.find("(v1)-[friend]->(v2); (v2)-[follow]->(v3)")  
  
# Filter the motifs (the content of the motifs DataFrame)  
# Select only the ones matching the pattern  
# vertex -> friend-> vertex -> follow ->vertex  
motifsFriendFollow = motifs\  
.filter("friend.relationship='friend' AND follow.relationship='follow' ")
```

To access a field of a struct column use the
syntax columnName.field

Basic statistics

- Some specific properties are provided to compute basic statistics on the degrees of the vertexes
 - degrees
 - inDegrees
 - outDegrees
- The returned result of each of this property is a DataFrame with
 - id
 - (in/out)Degree value

Basic statistics: degrees

- **degrees**
 - Returns the degree of each vertex
 - i.e., the number of edges associated with each vertex
 - The **result** is stored in a DataFrame with **Columns** (vertex) “**id**” and “**degree**”
 - One record per vertex
 - Only the vertexes with **degree>=1** are stored in the returned DataFrame

Basic statistics: inDegrees

- **inDegrees**
 - Returns the in-degree of each vertex
 - i.e., the number of in-edges associated with each vertex
 - The **result** is stored in a DataFrame with **Columns** (vertex) “**id**” and “**inDegree**”
 - One record per vertex
 - Only the vertexes with **in-degree ≥ 1** are stored in the returned DataFrame

Basic statistics: outDegrees

- **outDegrees**
 - Returns the out-degree of each vertex
 - i.e., the number of out-edges associated with each vertex
 - The **result** is stored in a DataFrame with **Columns** (vertex) “**id**” and “**outDegree**”
 - One record per vertex
 - Only the vertexes with **out-degree>=1** are stored in the returned DataFrame

Basic statistics: Example 1

- Given the input graph, compute
 - Degree of each vertex
 - inDegree of each vertex
 - outDegree of each vertex

Basic statistics: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Basic statistics: Example 1

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Basic statistics: Example 1

```
# Retrieve the DataFrame with the information about the degree of  
# each vertex  
vertexesDegreesDF = g.degrees
```

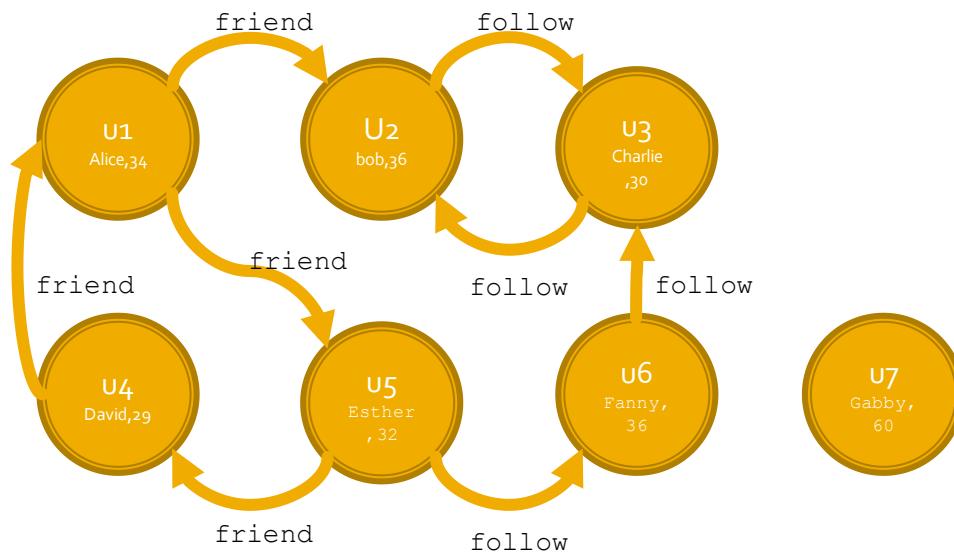
```
# Retrieve the DataFrame with the information about the in-degree of  
# each vertex  
vertexesInDegreesDF = g.inDegrees
```

```
# Retrieve the DataFrame with the information about the out-degree of  
# each vertex  
vertexesOutDegreesDF = g.outDegrees
```

Basic statistics: Example 2

- Given the input graph, select only the ids of the vertexes with at least 2 in-edges

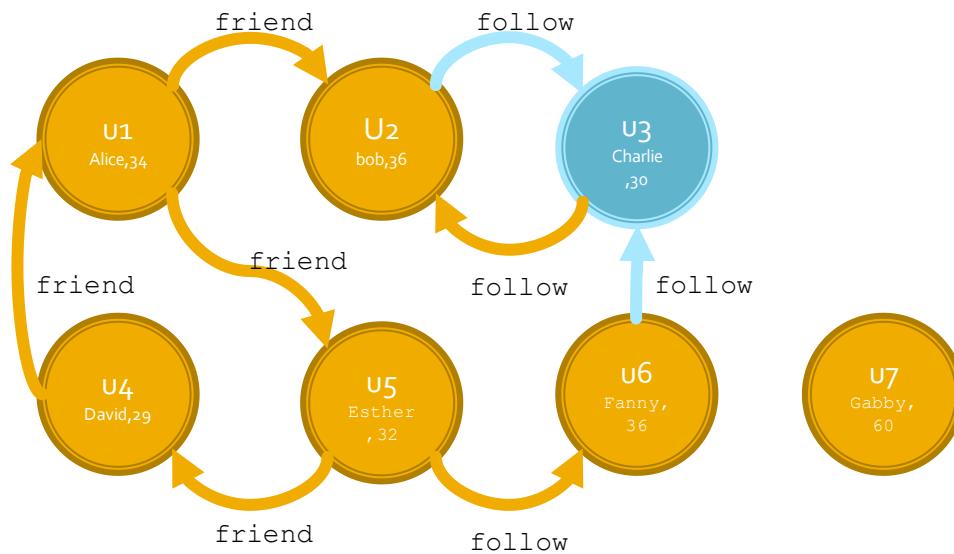
Input graph



Basic statistics: Example 2

- Given the input graph, select only the ids of the vertexes with at least 2 in-edges

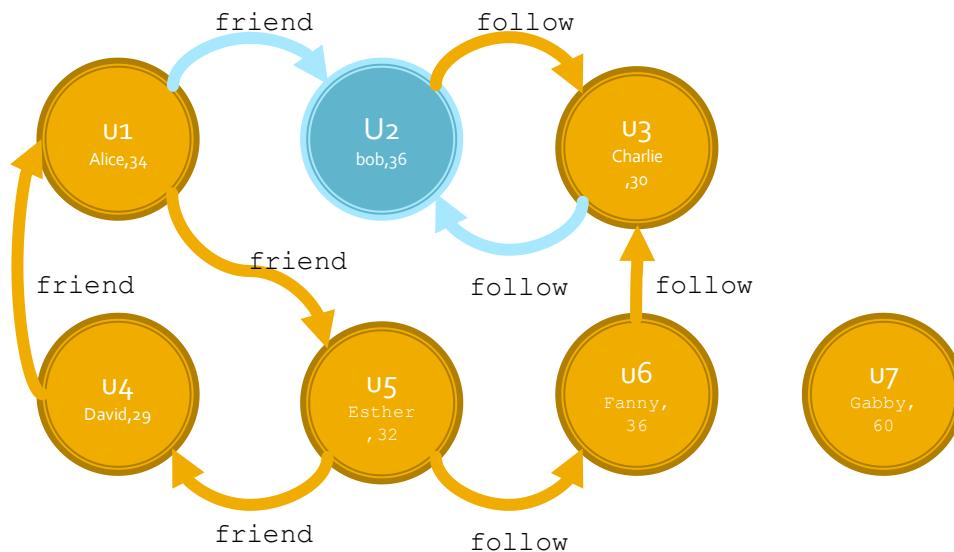
Input graph



Basic statistics: Example 2

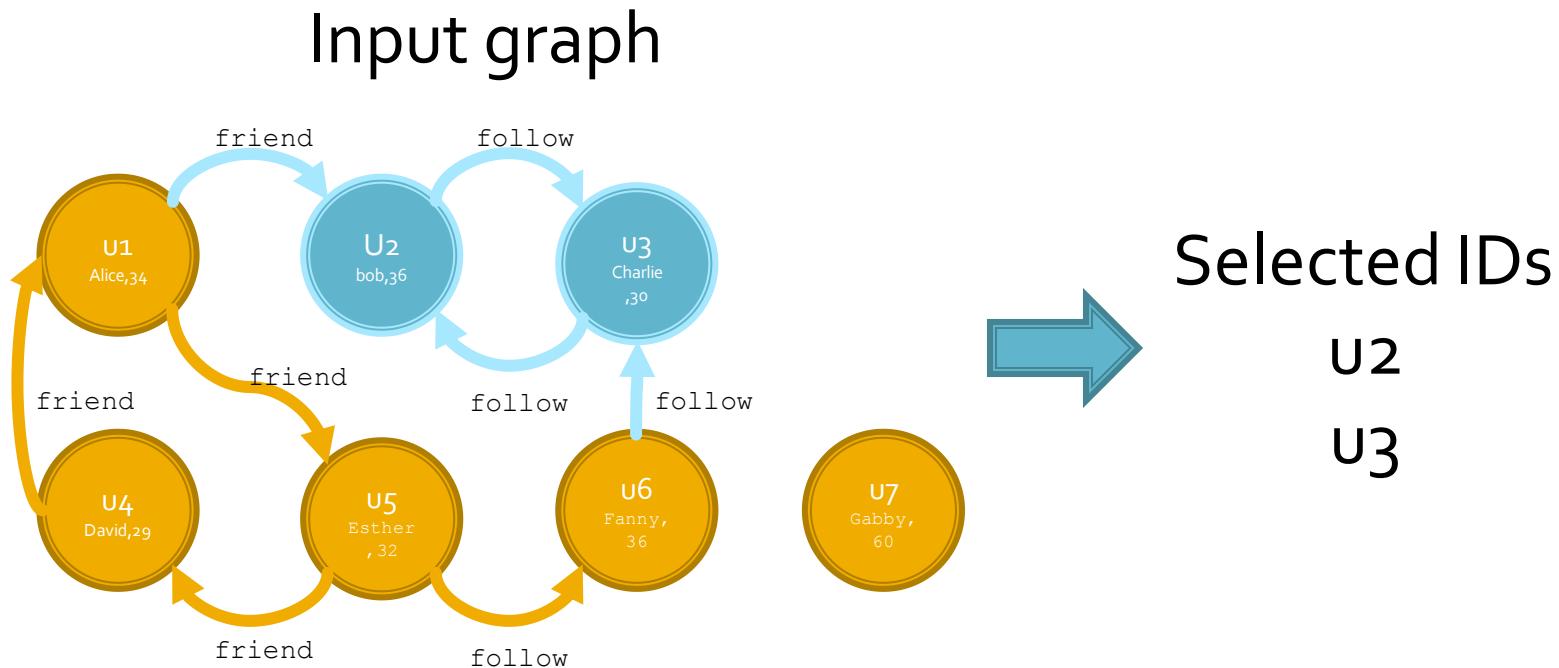
- Given the input graph, select only the ids of the vertexes with at least 2 in-edges

Input graph



Basic statistics: Example 2

- Given the input graph, select only the ids of the vertexes with at least 2 in-edges



Basic statistics: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Basic statistics: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Basic statistics: Example 2

```
# Retrieve the DataFrame with the information about the in-degree of  
# each vertex  
vertexesInDegreesDF = g.inDegrees  
  
# Select only the vertexes with and in-degree value >=2  
selectedVertexesDF = vertexesInDegreesDF.filter("inDegree>=2")  
  
# Select only the content of Column id  
selectedVertexesIDsDF = selectedVertexesDF.select("id")
```

Graph Analytics in Spark

Graph Algorithms with GraphFrames

Algorithms over graphs

- GraphFrame provides the parallel implementation of a set of state of the art algorithms for graph analytics
 - Breadth first search
 - Shortest paths
 - Connected components
 - Strongly connected component
 - Label propagation
 - PageRank
 - ...
- Custom algorithms can be designed and implemented

Checkpoint directory

- To run some expensive algorithms, set a checkpoint directory that will store the state of the job at every iteration
- This allow you to continue where you left off if the job crashes
- Create such a folder to set the checkpoint directory with:

`sc.setCheckpointDir(graphframes_ckpts_dir)`

- `graphframes_ckpts_dir` is your new checkpoint folder
- `sc` is your `SparkContext` object
 - Retrieve it from a `SparkSession` by using `spark.sparkContext`

Breadth first search

- Breadth-first search (BFS) is an algorithm for traversing/searching graph data structures
 - It finds the shortest path(s) from one vertex (or a set of vertexes) to another vertex (or a set of vertexes).
 - It is used in many other algorithms
 - Length of shortest paths
 - Connected components
 - ...

Breadth first search

- The `bfs(fromExpr, toExpr, edgeFilter=None maxPathLength=10)` method of the `GraphFrame` class returns the **shortest path(s)** from the vertexes matching expression `fromExpr` expression to vertexes matching expression `toExpr`
 - If there are many vertexes matching `fromExpr` and `toExpr`, only the couple(s) with the shortest length is returned

Breadth first search

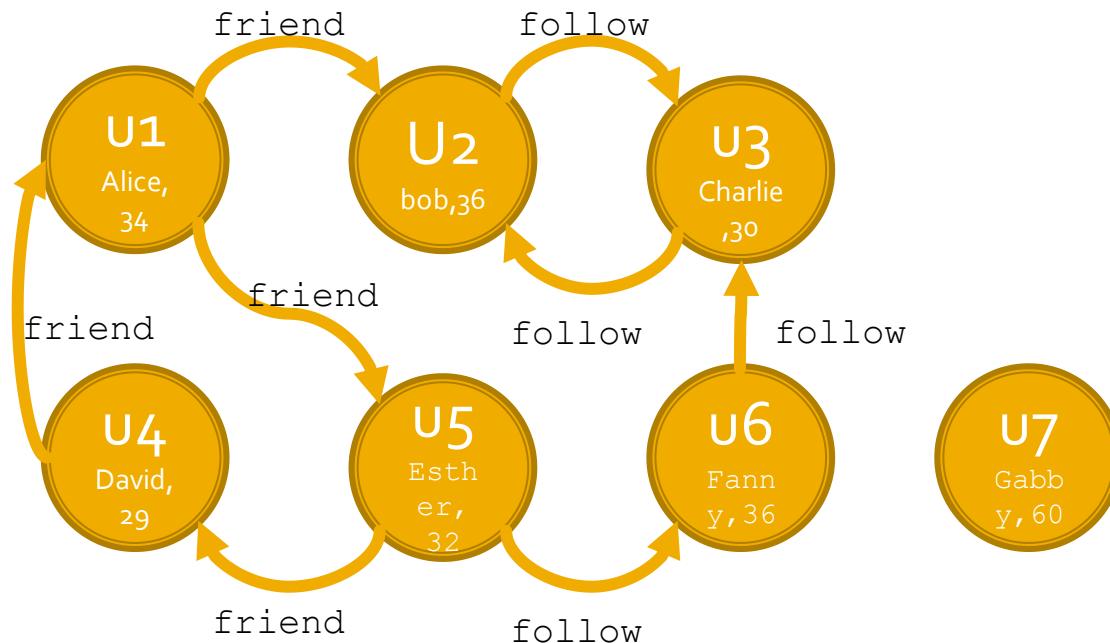
- **fromExpr**: Spark SQL expression specifying valid starting vertexes for the execution of the BFS algorithm
 - E.g., to start from a specific vertex
 - "id = [start vertex id]"
- **toExpr**: Spark SQL expression specifying valid target vertexes for the BFS algorithm
- **maxPathLength**: Limit on the length of paths (default = 10)
- **edgeFilter**: Spark SQL expression specifying edges that may be used in the search (default None)

Breadth first search

- `bfs()` returns a DataFrame containing the selected shortest path(s)
 - If `multiple paths` are valid and their `length` is equal to the shortest length, the returned DataFrame will contain `one Row for each path`
 - The `number of columns` of the returned DataFrame is equal to
 - `(length of the shortest path*2)+1`

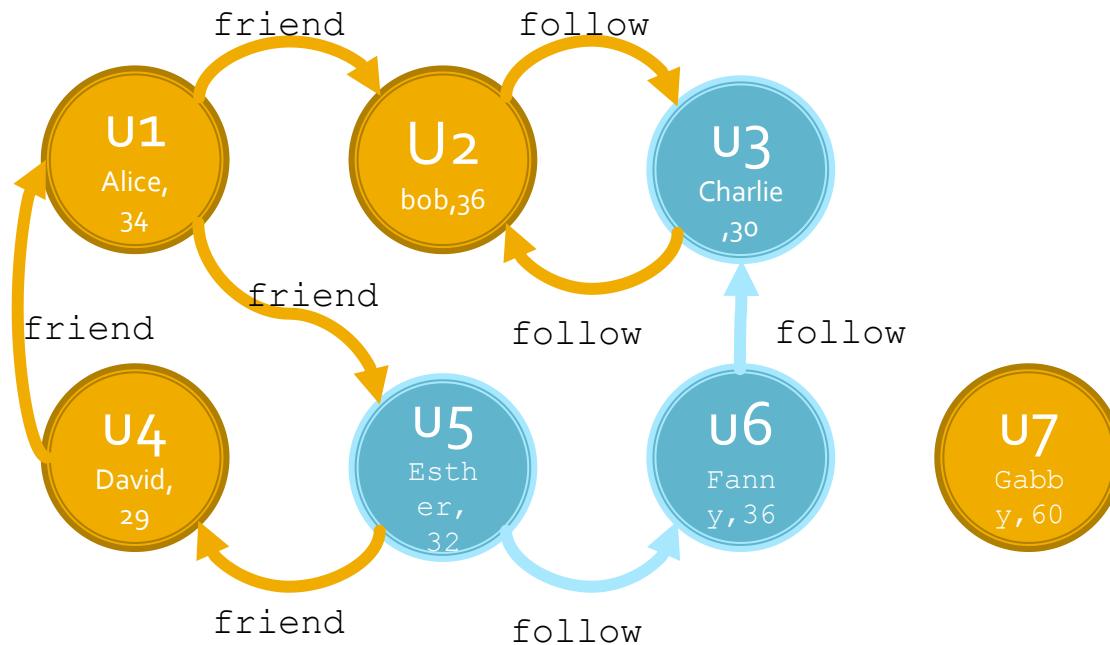
Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame



Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame



Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame

Content of the returned DataFrame

from	eo	v1	e1	to
[u5, Esther, 32]	[u5, u6, follow]	[u6, Fanny, 36]	[u6, u3, follow]	[u3, Charlie, 30]

Breadth first search: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 1

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

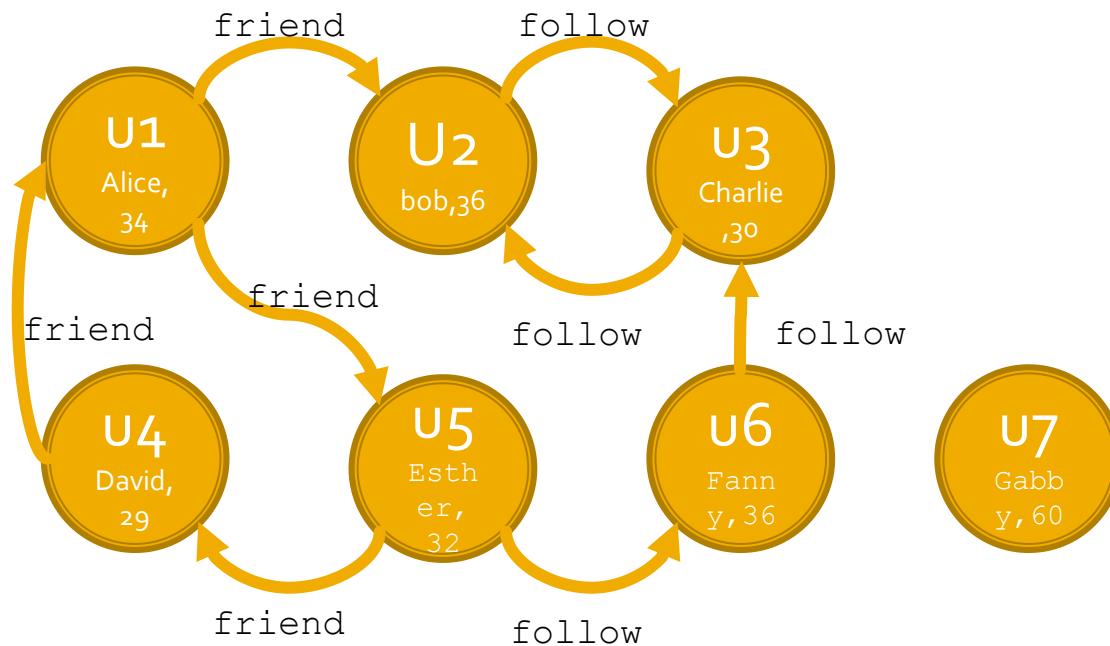
# Create the graph
g = GraphFrame(v, e)
```

Breadth first search: Example 1

```
# Search from vertex with name = "Esther" to vertex with name = "Charlie"  
  
shortestPaths = g.bfs("name = 'Esther' ", "name = 'Charlie' ")
```

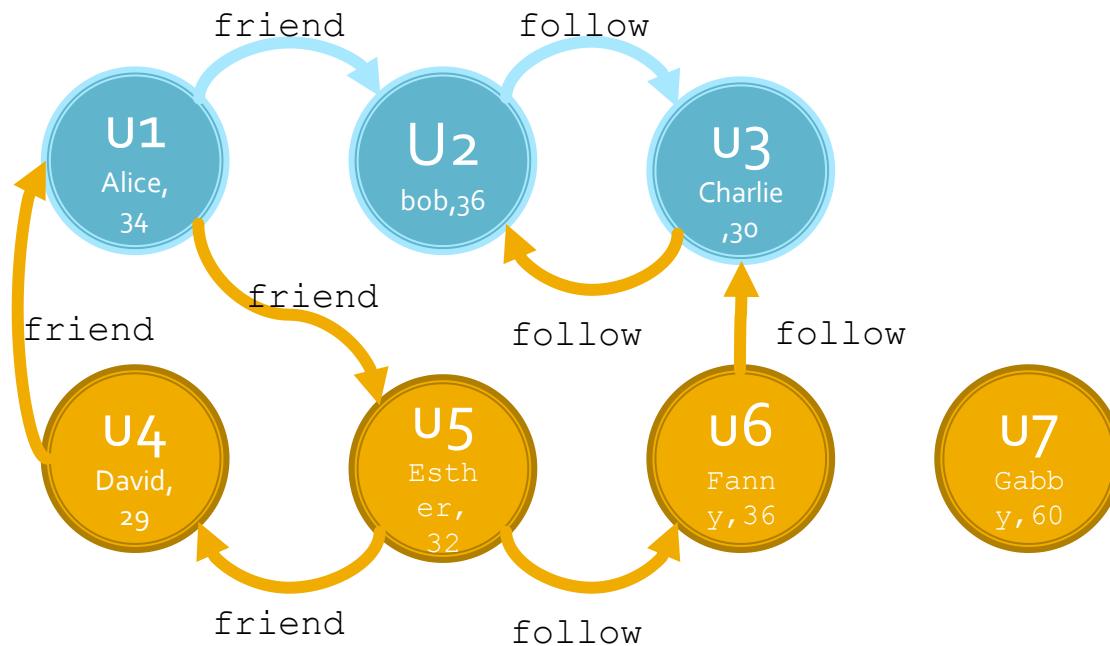
Breadth first search: Example 2

- Find the shortest path from Alice to a user who is 30 years old
- Store the result in a DataFrame



Breadth first search: Example 2

- Find the shortest path from Alice to a user who is 30 years old
- Store the result in a DataFrame



Breadth first search: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

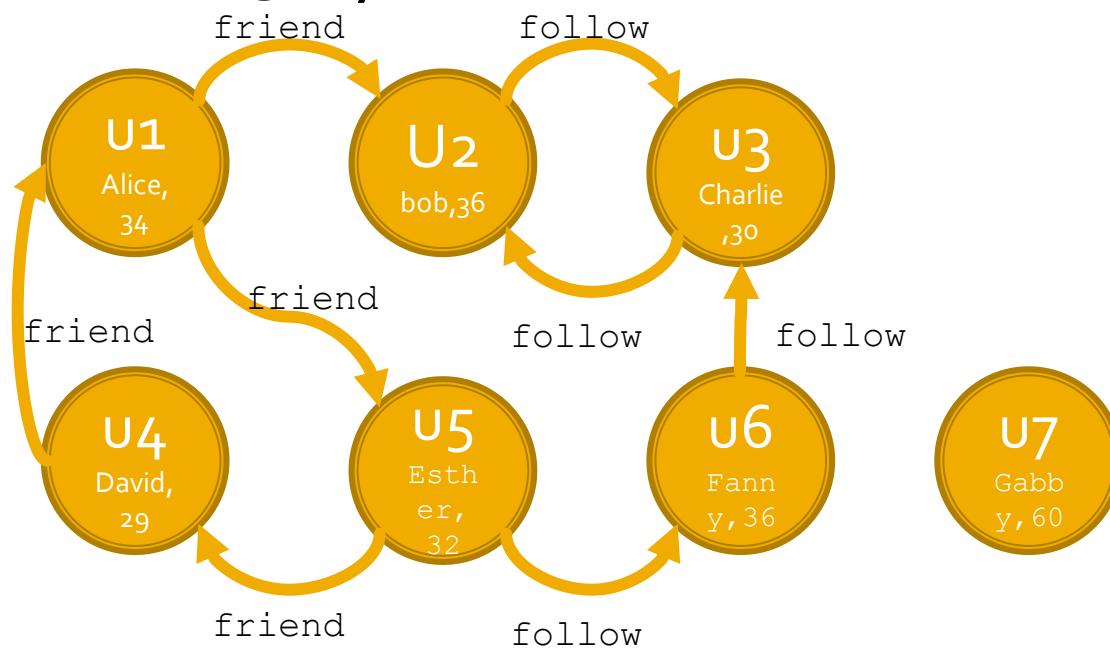
Breadth first search: Example 2

```
# Find the shortest path from Alice to a user who is 30 years old
```

```
shortestPaths = g.bfs("name = 'Alice'", "age= 30")
```

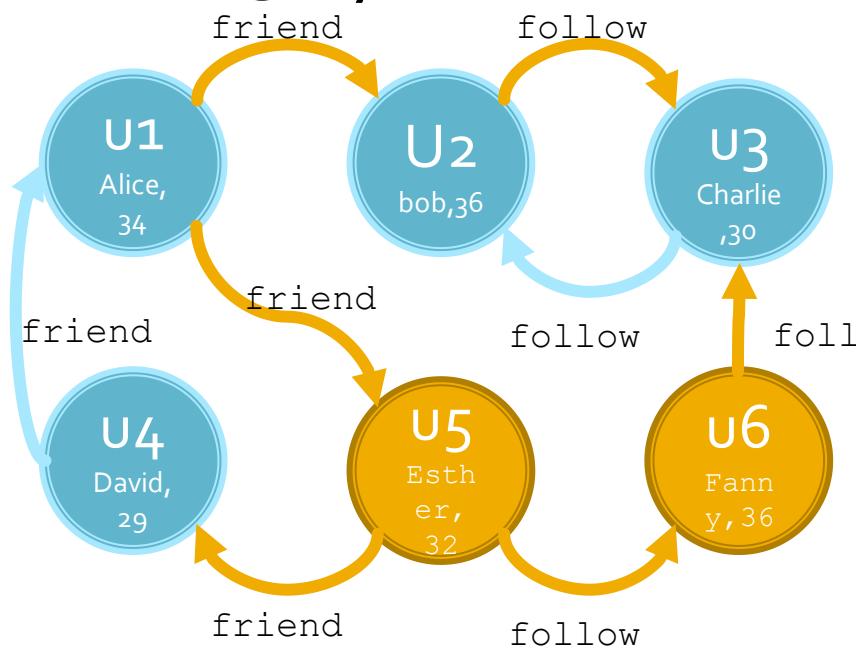
Breadth first search: Example 3

- Find the shortest path from any user who is less than 31 years old to any user who is more than 30 years old



Breadth first search: Example 3

- Find the shortest path from any user who is less than 31 years old to any user who is more than 30 years old



Two paths are selected in this case

Breadth first search: Example 3

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 3

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

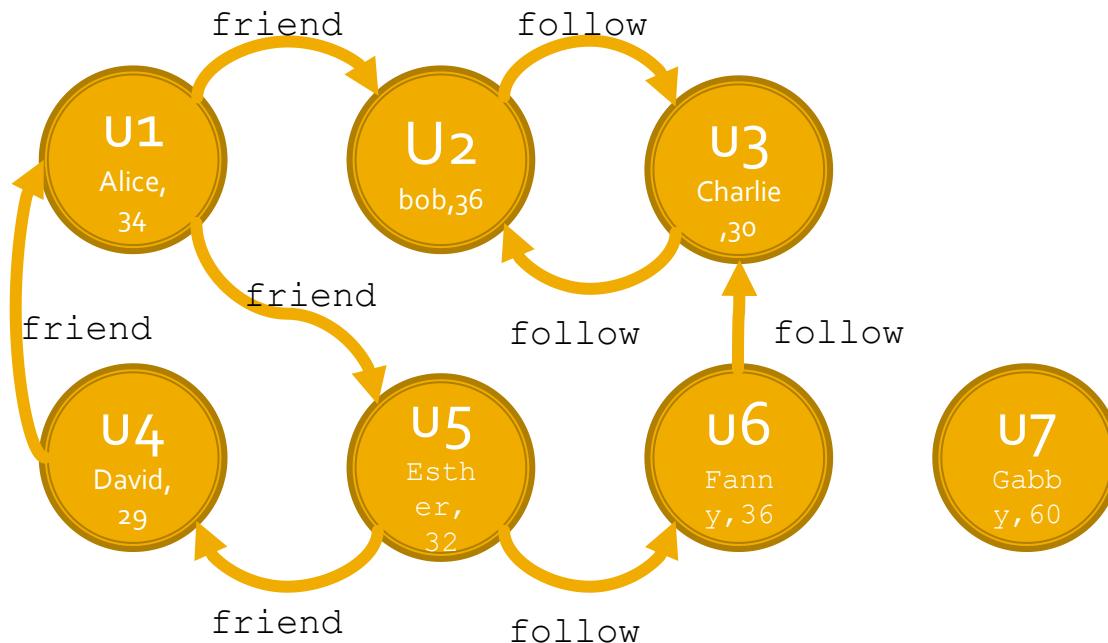
Breadth first search: Example 3

```
# Find the shortest path from any user who is less than 31 years old  
# to any user who is more than 30 years old
```

```
shortestPaths = g.bfs("age<31", "age>30")
```

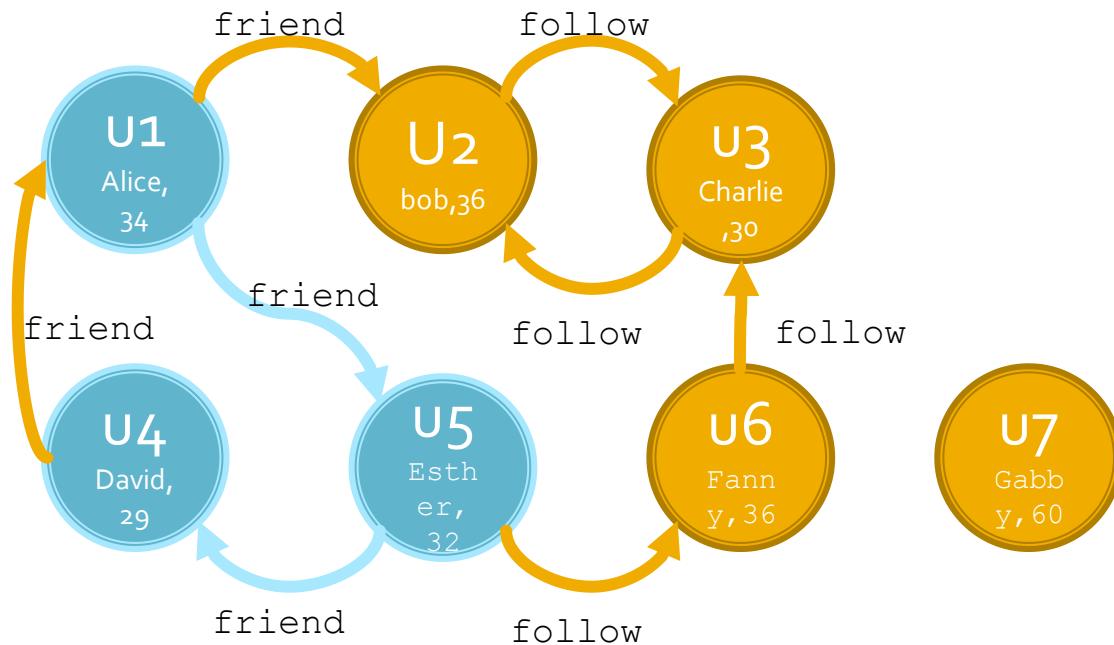
Breadth first search: Example 4

- Find the shortest path from Alice to any user who is less than 31 years old without using “follow” edges



Breadth first search: Example 4

- Find the shortest path from Alice to any user who is less than 31 years old without using “follow” edges



Breadth first search: Example 4

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 4

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Breadth first search: Example 4

```
# Find the shortest path from Alice to any user who is less  
# than 31 years old without using "follow" edges
```

```
shortestPaths = g.bfs("name = 'Alice'", "age<31", "relationship<>'follow' ")
```

Shortest path

- The shortest path method selects the length of the shortest path(s) from each vertex to a given set of landmark vertexes
 - It uses the BFS algorithm for computing the shortest paths

Shortest path

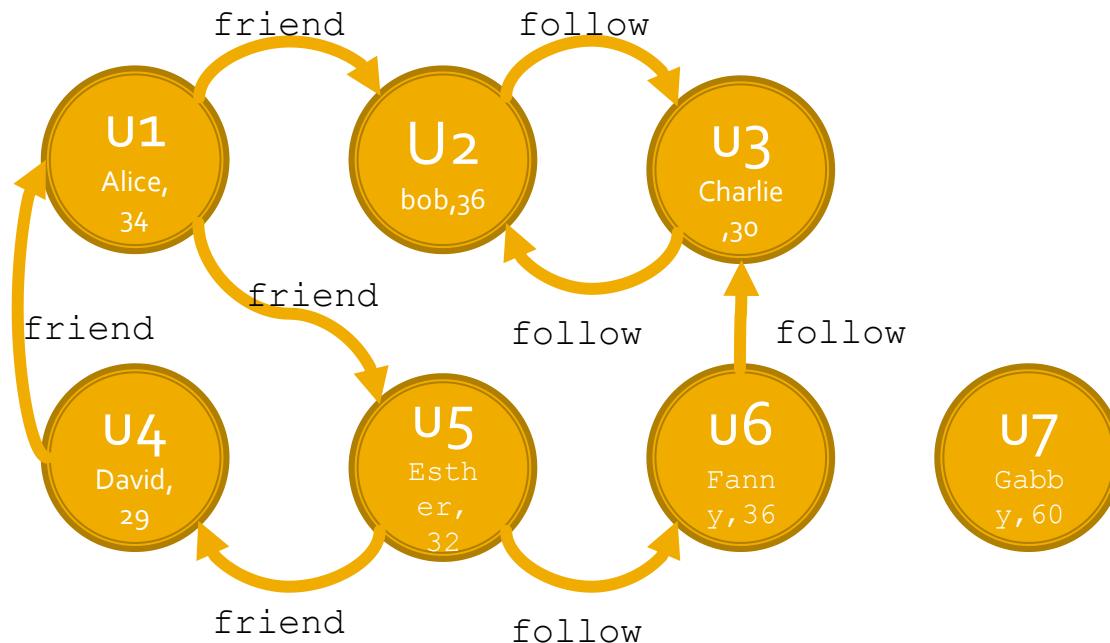
- The **shortestPaths(landmarks)** method of the GraphFrame class returns the **length of the shortest path(s)** from each vertex to a given set of **landmarks** vertexes
 - For each vertex, one shortest path for each landmark vertex is computed and its length is returned
 - **landmarks**: list of IDs of landmark vertexes
 - E.g., ['u1', 'u4']

Shortest path

- **shortestPaths()** returns a DataFrame that
 - Contains one record/Row for each distinct vertex of the input graph
 - Also for the non-connected ones
 - Is characterized by the following columns
 - One column for each attribute of the vertexes
 - **distances** (type **map**)
 - For each landmark **lm** it contains one pair (ID **lm**: length shortest path from the vertex of the current record to **lm**)

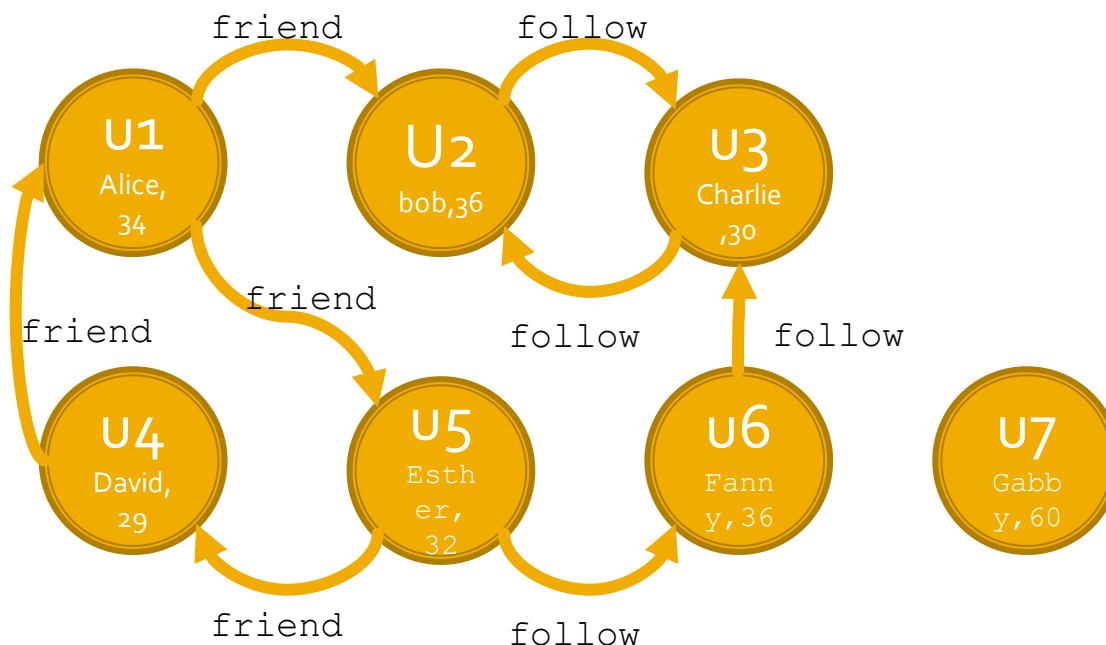
Shortest path: Example 1

- Find for each user the length of the shortest path to user u_1 (i.e., Alice)



Shortest path: Example 1

- Find for each user the length of the shortest path to user u_1 (i.e., Alice)



Vertex	Distance to u_1
u_1	0
u_2	-
u_3	-
u_4	1
u_5	2
u_6	-
u_7	-

Shortest path: Example 1

- Find for each user the length of the shortest path to user u_1 (i.e., Alice)

Content of the returned DataFrame

+-----+ id +-----+	+-----+ name +-----+	+-----+ age +-----+	+-----+ distances +-----+
u_1 Alice 34 $[u_1 \rightarrow 0]$			
u_2 Bob 36 []			
u_3 Charlie 30 []			
u_4 David 29 $[u_1 \rightarrow 1]$			
u_5 Esther 32 $[u_1 \rightarrow 2]$			
u_6 Fanny 36 []			
u_7 Gabby 60 []			

Shortest path: Example 1

- Find for each user the length of the shortest path to user u1 (i.e., Alice)

Content of the returned DataFrame

+-----+ <th> id name age distances <th>+-----+</th></th>	id name age distances <th>+-----+</th>	+-----+	
id name age distances <th>+-----+<th> </th></th>	+-----+ <th> </th>		
u1 Alice 34 [u1 -> 0]			
u2 Bob 36 []			
u3 Charlie 30 []			
u4 David 29 [u1 -> 1]			
u5 Esther 32 [u1 -> 2]			
u6 Fanny 36 []			
u7 Gabby 60 []			
+-----+ <th> </th> <th> </th> <th> </th>			

Data type: map
- It stores a set of pairs (Key: Value)

Shortest path: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Shortest path: Example 1

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

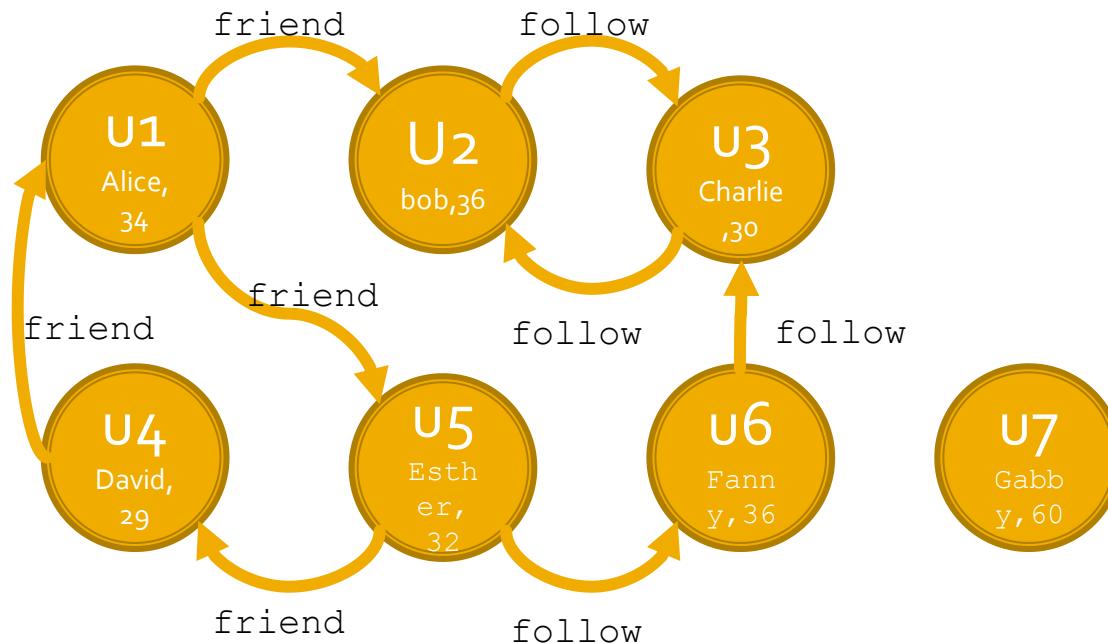
Shortest path: Example 1

```
# Find for each user the length of the shortest path to user u1
```

```
shortestPaths = g.shortestPaths(["u1"])
```

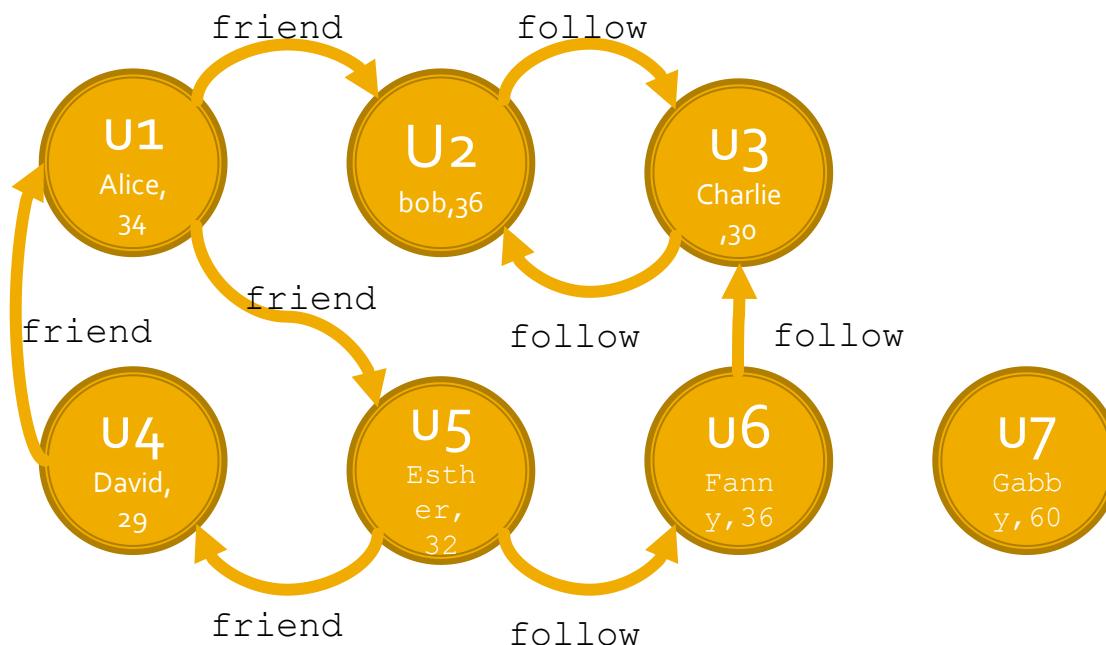
Shortest path: Example 2

- Find for each user the length of the shortest path to users **u₁** (Alice) and **u₄** (David)



Shortest path: Example 2

- Find for each user the length of the shortest path to users u_1 (Alice) and u_4 (David)



Vertex	Distance to u_1	Distance to u_4
u_1	0	2
u_2	-	-
u_3	-	-
u_4	1	0
u_5	2	1
u_6	-	-
u_7	-	-

Shortest path: Example 2

- Find for each user the length of the shortest path to users u_1 (Alice) and u_4 (David)

Content of the returned DataFrame

+-----+ id +-----+	+-----+ name +-----+	+-----+ age +-----+	+-----+ distances +-----+	
u_1 Alice 34 $[u_1 \rightarrow 0, u_4 \rightarrow 2]$				
u_2 Bob 36 []				
u_3 Charlie 30 []				
u_4 David 29 $[u_1 \rightarrow 1, u_4 \rightarrow 0]$				
u_5 Esther 32 $[u_1 \rightarrow 2, u_4 \rightarrow 1]$				filter('distance.u4<2')
u_6 Fanny 36 []				
u_7 Gabby 60 []				

Shortest path: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Shortest path: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Shortest path: Example 2

```
# Find for each user the length of the shortest paths to users u1 and u4
```

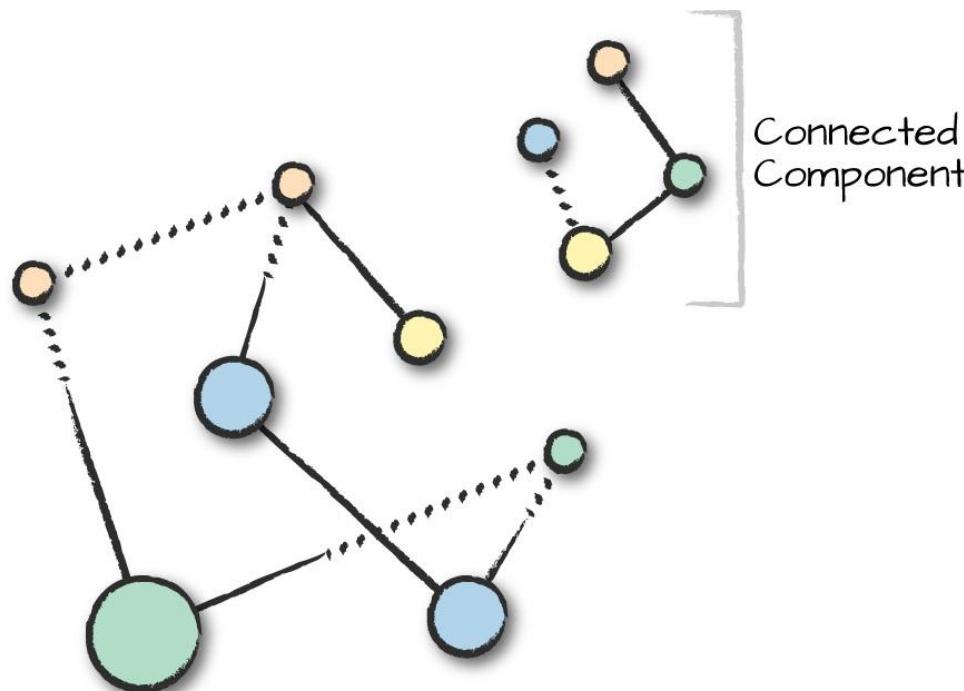
```
shortestPaths = g.shortestPaths(["u1", "u4"])
```

Connected components

- A connected component of a graph is a subgraph **sg** such that
 - Any two vertexes in **sg** are connected to each other by at least one path
 - The set of vertexes in **sg** is not connected to any additional vertexes in the original graph
 - Direction of edges is not considered

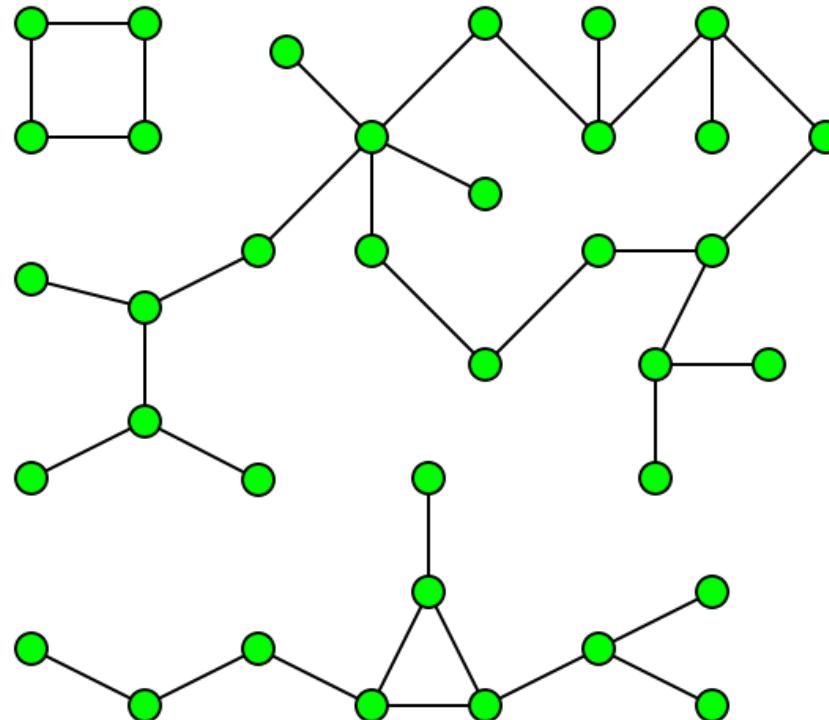
Connected components

- Two connected components



Connected components

- Three connected components



Connected components

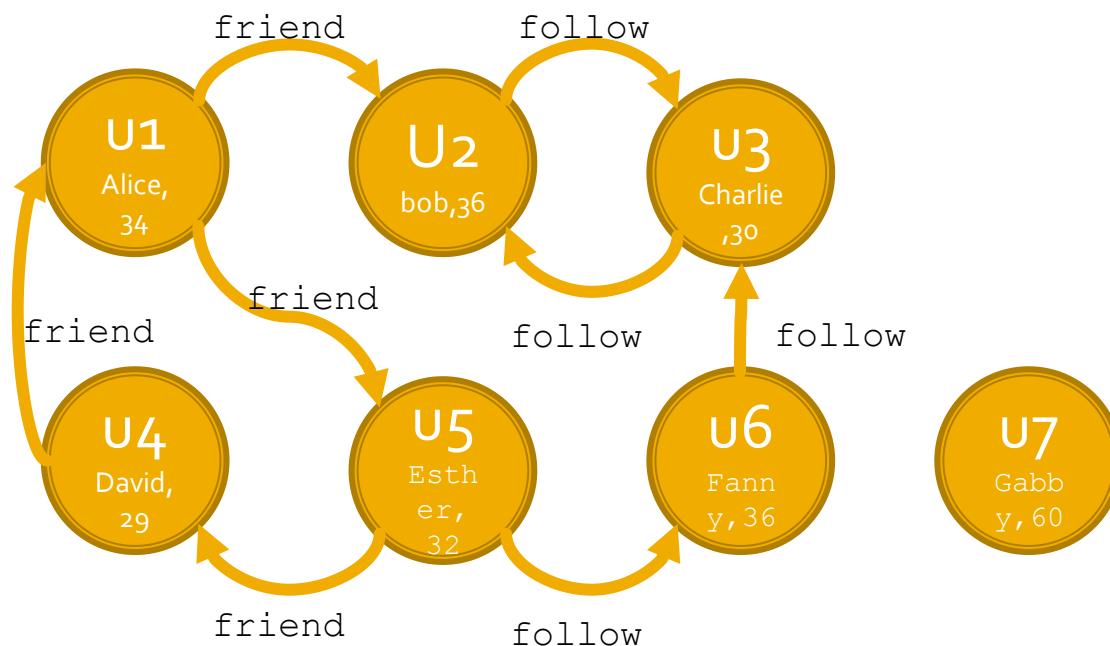
- The `connectedComponents()` method of the `GraphFrame` class returns the connected components of the input graph
 - It is an expensive algorithm
 - It requires setting a Spark checkpoint directory

Connected components

- **connectedComponents()** returns a DataFrame that
 - Contains one record/Row for each distinct vertex of the input graph
 - Is characterized by the following columns
 - One column for each attribute of the vertexes
 - **component** (type **long**)
 - It is the identifier of the connected component to which the current vertex has been assigned

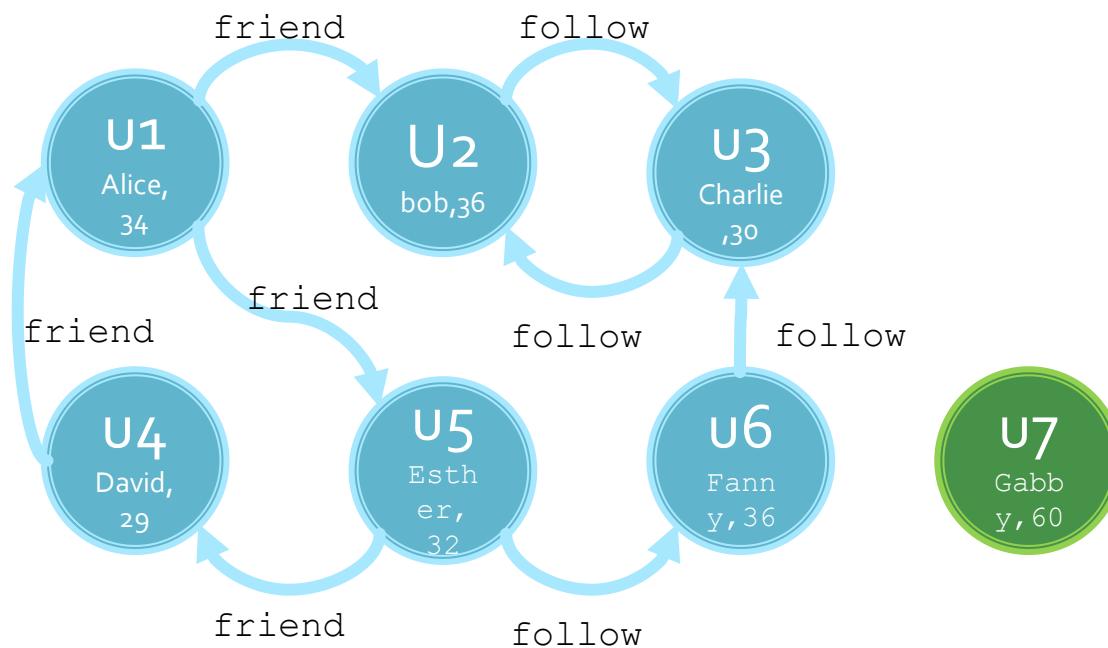
Connected components: Example

- Print on the stdout the number of connected components of the following graph



Connected components: Example

- Print on the stdout the number of connected components of the following graph



The are two connected components on this graph

Connected components: Example

- Print on the stdout the number of connected components of the following graph

Content of the DataFrame used to store the two identified connected components

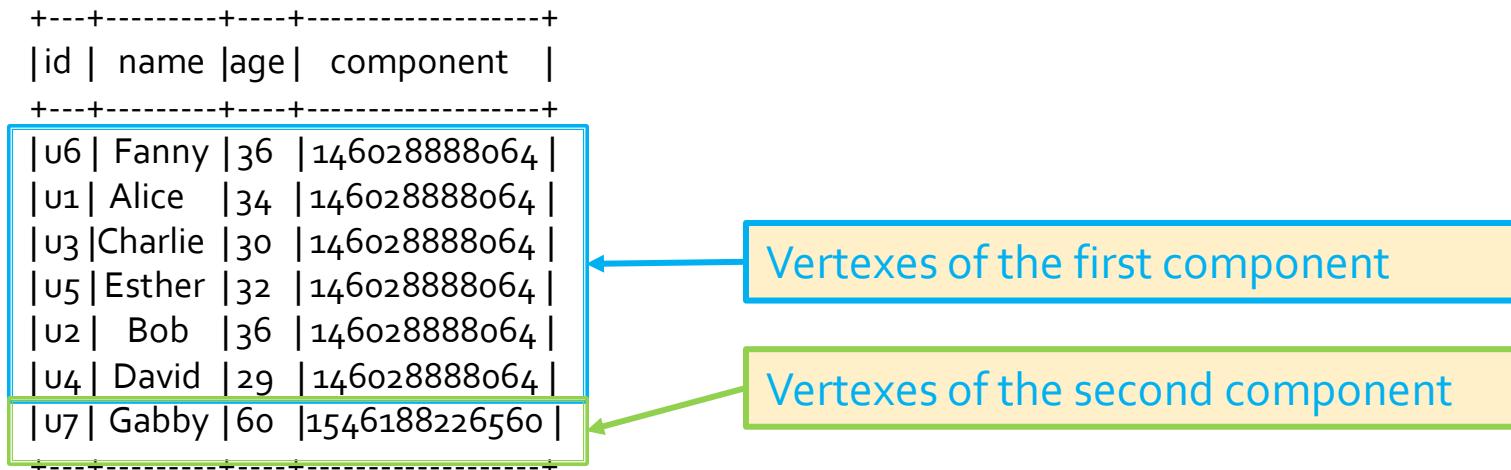
+-----+ id	+-----+ name	+-----+ age	+-----+ component
u6 Fanny 36 146028888064			
u1 Alice 34 146028888064			
u3 Charlie 30 146028888064			
u5 Esther 32 146028888064			
u2 Bob 36 146028888064			
u4 David 29 146028888064			
u7 Gabby 60 1546188226560			

Connected components: Example

- Print on the stdout the number of connected components of the following graph

Content of the DataFrame used to store the two identified connected components

+-----+ id +-----+	+-----+ name +-----+	+-----+ age +-----+	+-----+ component +-----+
u6 Fanny 36 146028888064			
u1 Alice 34 146028888064			
u3 Charlie 30 146028888064			
u5 Esther 32 146028888064			
u2 Bob 36 146028888064			
u4 David 29 146028888064			
u7 Gabby 60 1546188226560			


Vertexes of the first component
Vertexes of the second component

Connected components: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Connected components: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Connected components: Example

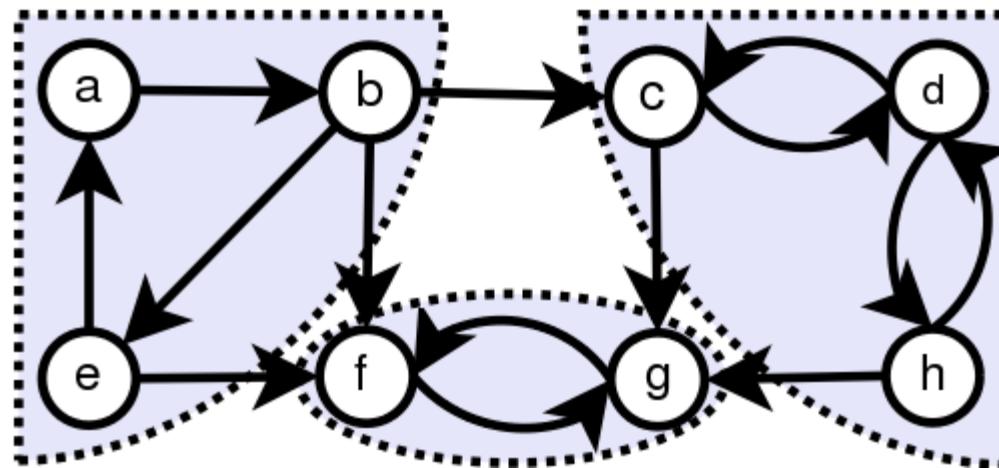
```
# Set checkpoint folder  
sc.setCheckpointDir("tmp_ckpts")  
  
# Run the algorithm  
connComp=g.connectedComponents()  
  
# Count the number of components  
nComp=connComp.select("component").distinct().count()  
  
print("Number of connected components: ", nComp)
```

Strongly Connected components

- A directed subgraph sg is called strongly connected if every vertex in sg is reachable from every other vertex in sg
 - For undirected graph, connected and strongly connected components are the same

Strongly Connected components

- A graph with 3 strongly connected subgraphs/components



Strongly Connected components

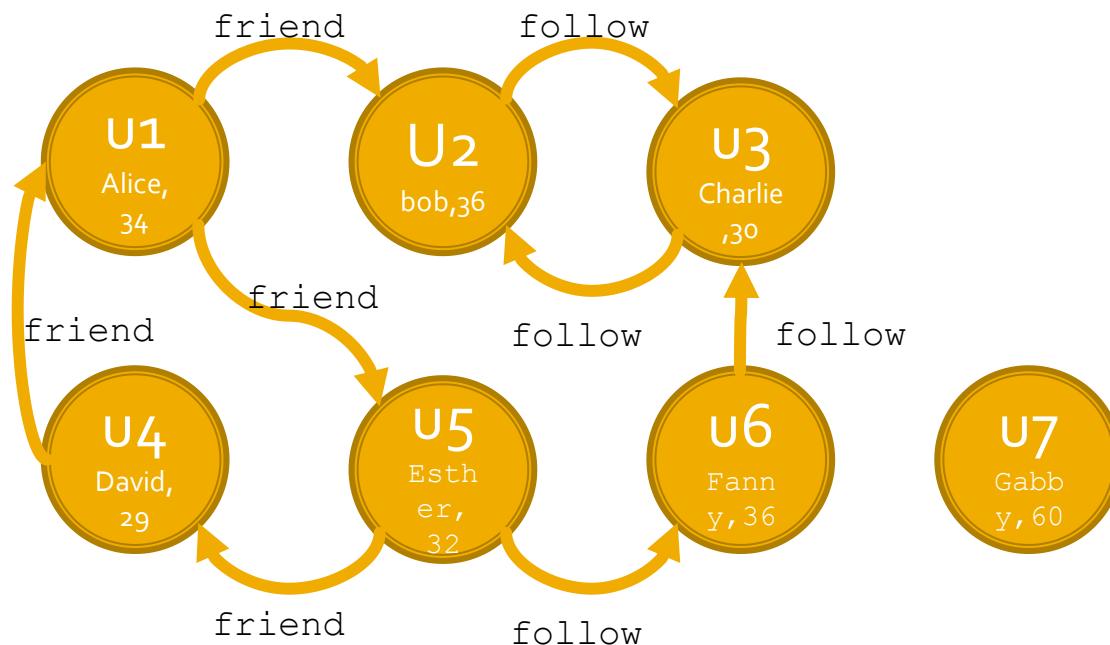
- The `stronglyConnectedComponents()` method of the `GraphFrame` class returns the strongly connected components of the input graph
 - It is an **expensive algorithm**
 - Better to run on a cluster with yarn scheduler even with small graphs
 - It requires setting a Spark **checkpoint directory**

Strongly Connected components

- **stronglyConnectedComponents()** returns a DataFrame that
 - Contains one record/Row for each distinct vertex of the input graph
 - Is characterized by the following columns
 - One column for each attribute of the vertexes
 - component (type long)
 - It is the identifier of the strongly connected component to which the current vertex has been assigned

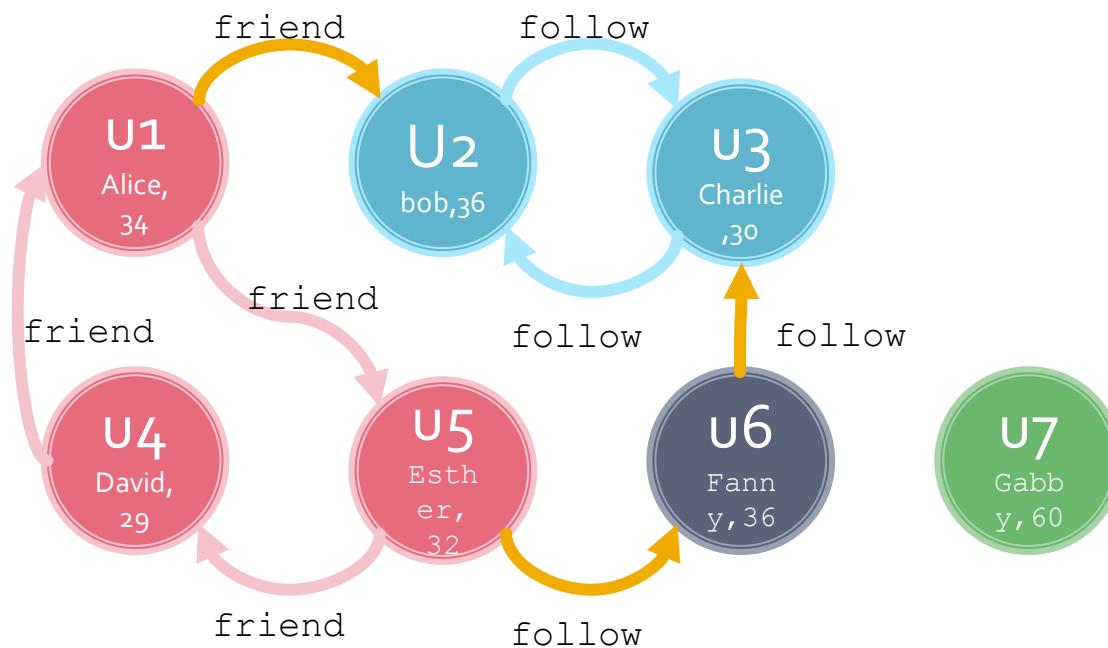
Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph



Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph



The are four connected components on this graph

Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph

Content of the DataFrame used to store the identified strongly connected components

+-----+ id	+-----+ name	+-----+ age	+-----+ component	+-----+
u3 Charlie	30	146028888064		
u2 Bob	36	146028888064		
u1 Alice	34	498216206336		
u5 Esther	32	498216206336		
u4 David	29	498216206336		
u6 Fanny	36	1090921693184		
u7 Gabby	60	1546188226560		

Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph

Content of the DataFrame used to store the identified strongly connected components

+-----+ id +-----+	+-----+ name +-----+	+-----+ age +-----+	+-----+ component +-----+
u3 Charlie 30 146028888064			
u2 Bob 36 146028888064			
u1 Alice 34 498216206336			
u5 Esther 32 498216206336			
u4 David 29 498216206336			
u6 Fanny 36 1090921693184			
u7 Gabby 60 1546188226560			

Vertexes of the first SCC

Vertexes of the second SCC

Vertexes of the third component

Vertexes of the fourth component

Strongly Connected components: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Strongly Connected components: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Strongly Connected components: Example

```
# Set checkpoint folder  
sc.setCheckpointDir("tmp_ckpts")  
  
# Run the algorithm  
strongConnComp = g.stronglyConnectedComponents(maxIter=10)  
  
# Count the number of strongly connected components  
nComp=strongConnComp.select("component").distinct().count()  
  
print("Number of strongly connected components: ", nComp)
```

Label propagation

- Label Propagation is an algorithm for detecting communities in graphs
 - Like clustering but exploiting connectivity
 - **Convergence is not guaranteed**
 - One can end up with trivial solutions

Label propagation

- The Label Propagation algorithm
 - Each vertex in the network is initially assigned to its own community
 - At every step, vertexes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages

Label propagation

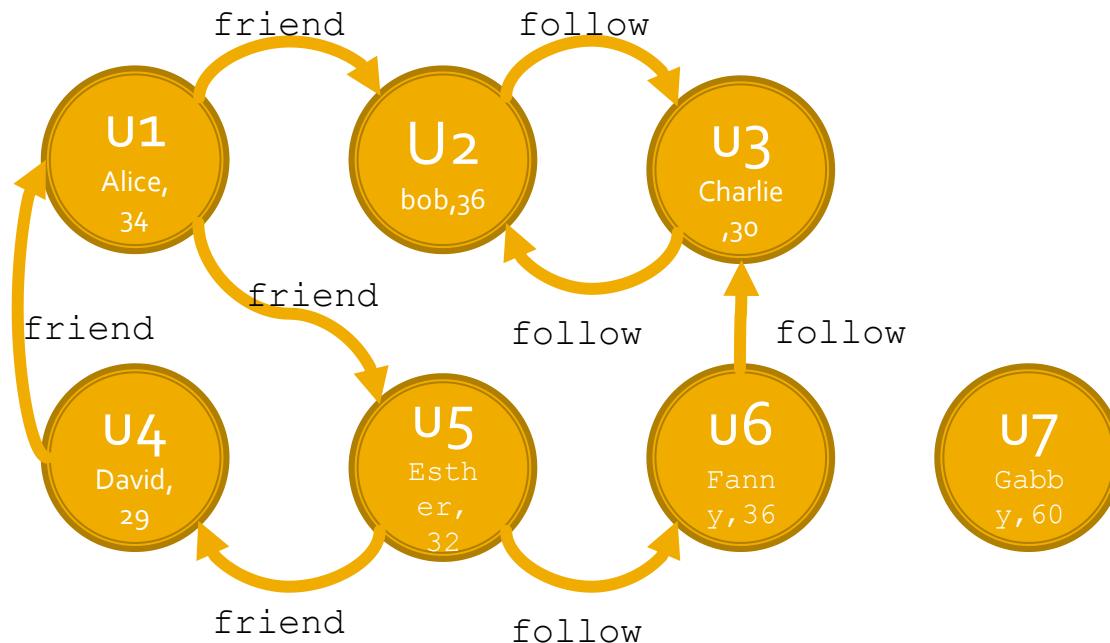
- The **labelPropagation(maxIter)** method of the GraphFrame class runs and returns the result of the label propagation algorithm
 - Parameter maxIter:
 - The number of iterations to run

Label propagation

- **labelPropagation()** returns a DataFrame that
 - Contains one record/Row for each distinct vertex of the input graph
 - Is characterized by the following columns
 - One column for each attribute of the vertexes
 - label (type long)
 - It is the identifier of the community to which the current vertex has been assigned

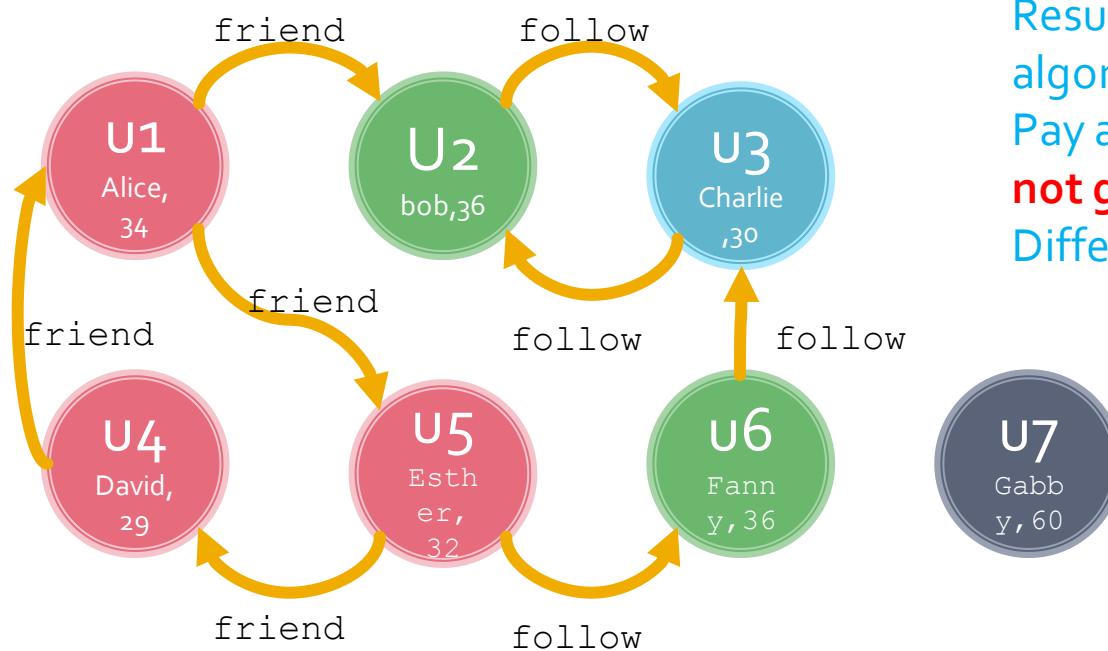
Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm



Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm



Result returned by one run of the algorithm.
Pay attention that **convergence is not guarantee**.
Different results for different runs.

Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm

Content of the DataFrame used to store the identified communities

+-----+ id	+-----+ name	+-----+ age	+-----+ label	+-----+
u3 Charlie	30	146028888064		
u4 David	29	498216206336		
u1 Alice	34	498216206336		
u5 Esther	32	498216206337		
u7 Gabby	60	1546188226560		
u2 Bob	36	1606317768704		
u6 Fanny	36	1606317768704		

Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm

Content of the DataFrame used to store the identified communities

+-----+ id +-----+	+-----+ name +-----+	+-----+ age +-----+	+-----+ label +-----+
u3 Charlie 30 146028888064			
u4 David 29 498216206336			
u1 Alice 34 498216206336			
u5 Esther 32 498216206337			
u7 Gabby 60 1546188226560			
u2 Bob 36 1606317768704			
u6 Fanny 36 1606317768704			

Vertexes of the first community

Vertexes of the second community

Vertexes of the third community

Vertexes of the fourth community

Label propagation: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Label propagation: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Label propagation: Example

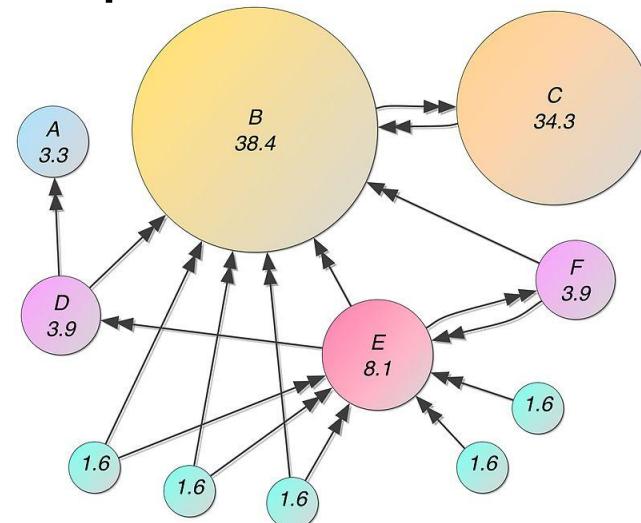
```
# Run the label propagation algorithm  
labelComm = g.labelPropagation(10)
```

PageRank

- PageRank is the original famous algorithm used by the Google Search engine to rank vertexes (web pages) in a graph by order of importance
 - For the Google search engine
 - Vertexes are web pages in the World Wide Web,
 - Edges are hyperlinks among web pages
 - It assigns a numerical weighting (importance) to each node

PageRank

- It computes a likelihood that a person randomly clicking on links will arrive at any particular web page
- For a high PageRank, it is important to
 - Have many in-links
 - Be liked by relevant pages (pages characterized by a high PageRank)



PageRank

- Basic idea
 - Each link's vote is proportional to the importance of its source page p
 - If page p with importance $\text{PageRank}(p)$ has n out-links, each out-link gets $\text{PageRank}(p)/n$ votes
 - Page p 's importance is the sum of the votes on its in-links

PageRank: Simple recursive formulation

1. # Initialize each page's rank to 1.0
For each p in pages set $\text{PageRank}(p)$ to 1.0
2. Iterate for max iterations
 - a. Page p sends a contribution $\text{PageRank}(p)/\text{numOutLinks}(p)$ to its neighbors (the pages it links)
 - b. Update each page's rank $\text{PageRank}(p)$ to $\text{sum}(\text{received contributions})$
 - c. Go to step 2

PageRank with Random jumps

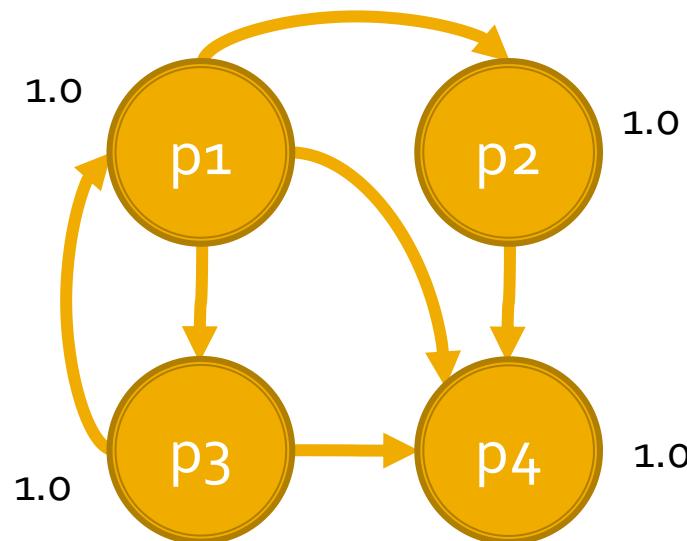
- The PageRank algorithm simulates the random walk of a user on the web
- At each step of the random walk, the random surfer has two options:
 - With probability $1-\alpha$, follow a link at random among the ones in the current page
 - With probability α , jump to a random page

PageRank with Random jumps

1. # Initialize each page's rank to 1.0
For each p in pages set $\text{PageRank}(p)$ to 1.0
2. Iterate for max iterations
 - a. Page p sends a contribution $\text{PageRank}(p)/\text{numOutLinks}(p)$ to its neighbors (the pages it links)
 - b. Update each page's rank $\text{PageRank}(p)$ to $\alpha + (1 - \alpha) * \text{sum}(\text{received contributions})$
 - c. Go to step 2

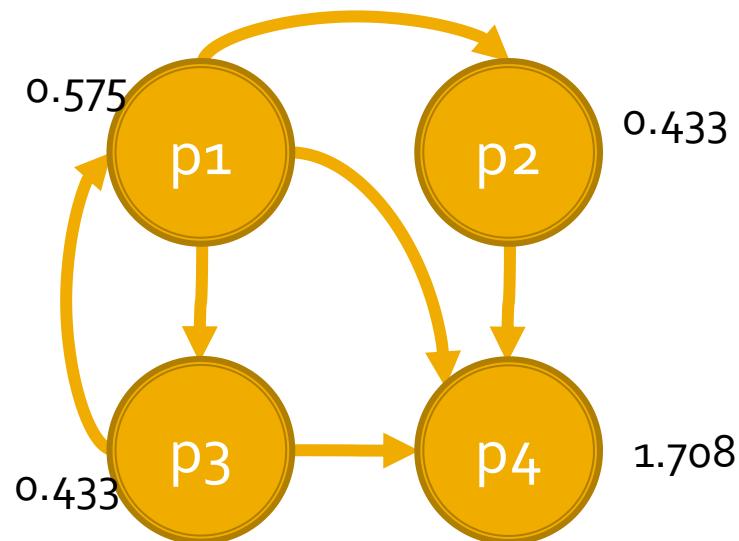
PageRank: Example

- $\alpha = 0.15$
- Initialization: $\text{PageRank}(p) = 1.0 \forall p$



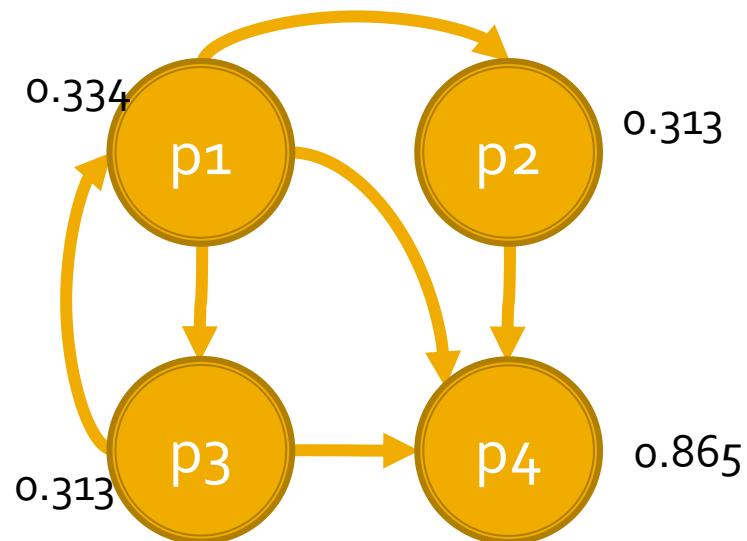
PageRank: Example

- Iteration #1



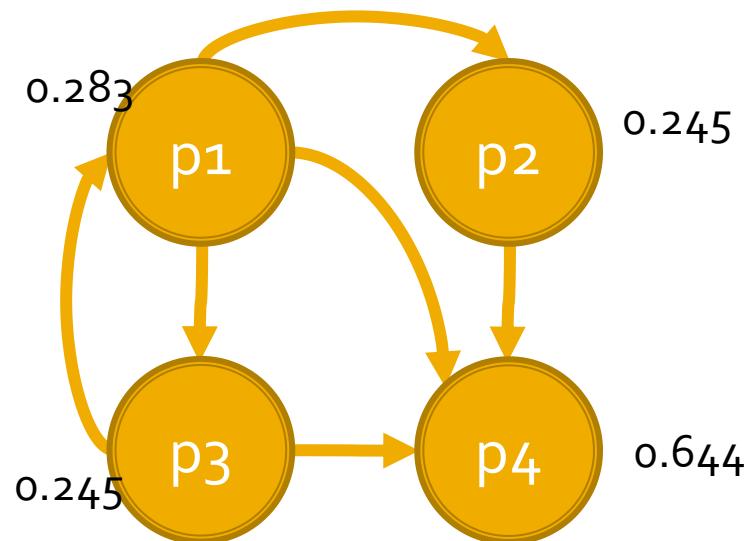
PageRank: Example

- Iteration #2



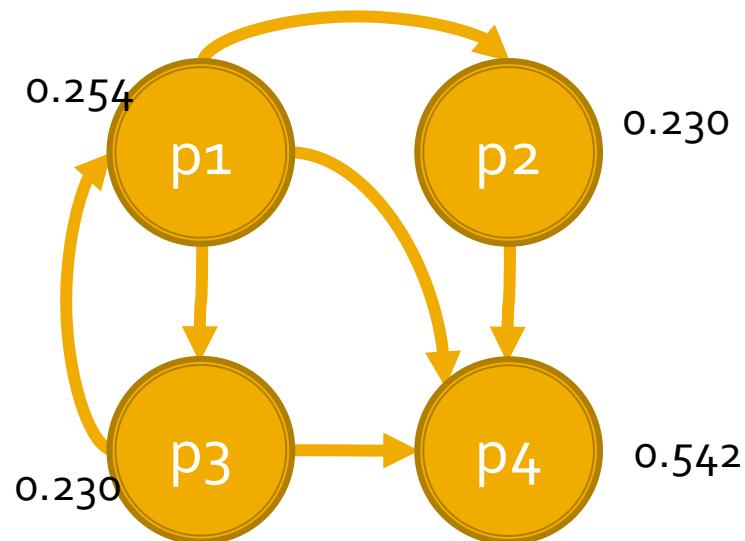
PageRank: Example

- Iteration #3



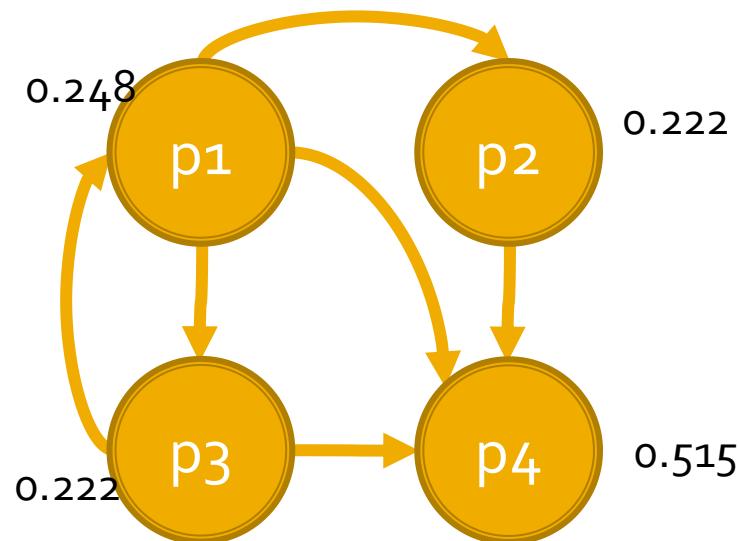
PageRank: Example

- Iteration #4



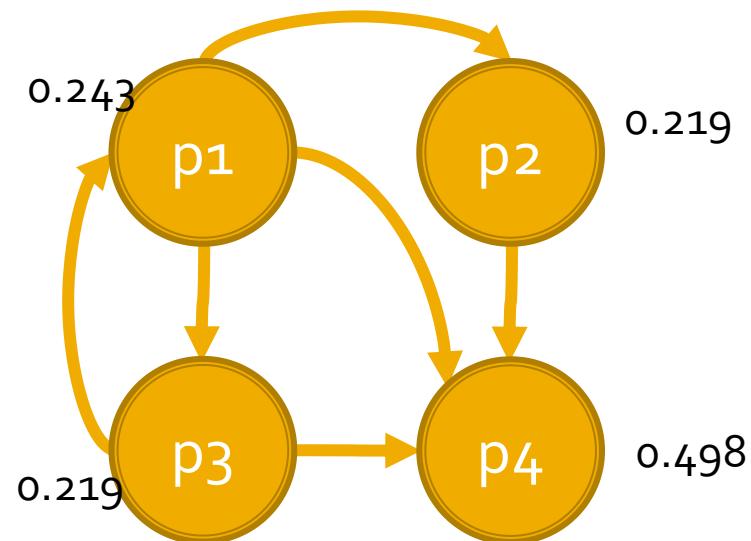
PageRank: Example

- Iteration #5



PageRank: Example

- Iteration #50



PageRank

- The `pageRank()` method of the `GraphFrame` class runs the PageRank algorithm on the input graph

PageRank

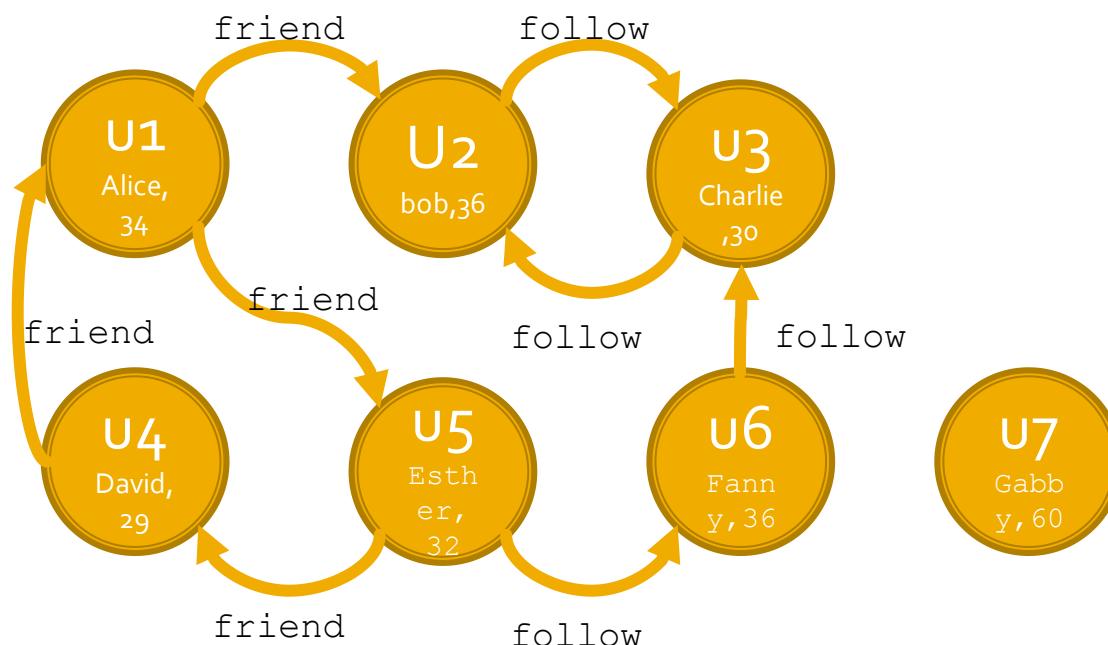
- Parameters
 - resetProbability
 - Probability of resetting to a random vertex (probability α associated with random jumps)
 - maxIter
 - If set, the algorithm is run for a fixed number of iterations
 - This may not be set if the tol parameter is set
 - Tol
 - If set, the algorithm is run until the given tolerance
 - This may not be set if the numIter parameter is set
 - sourceld (optional)
 - The source vertex for a personalized PageRank

PageRank

- `pageRank()` returns a new `GraphFrame` that
 - Contains the same vertexes and edges of the input graph
 - All the `vertexes` of the `new graph` are characterized by one `new attribute`, called “`pagerank`”, that stores the PageRank of the vertexes
 - The `edges` of the `new graph` are characterized by one `new attribute`, called “`weight`”, that stores the weight (PageRank contribution) propagated through that edge

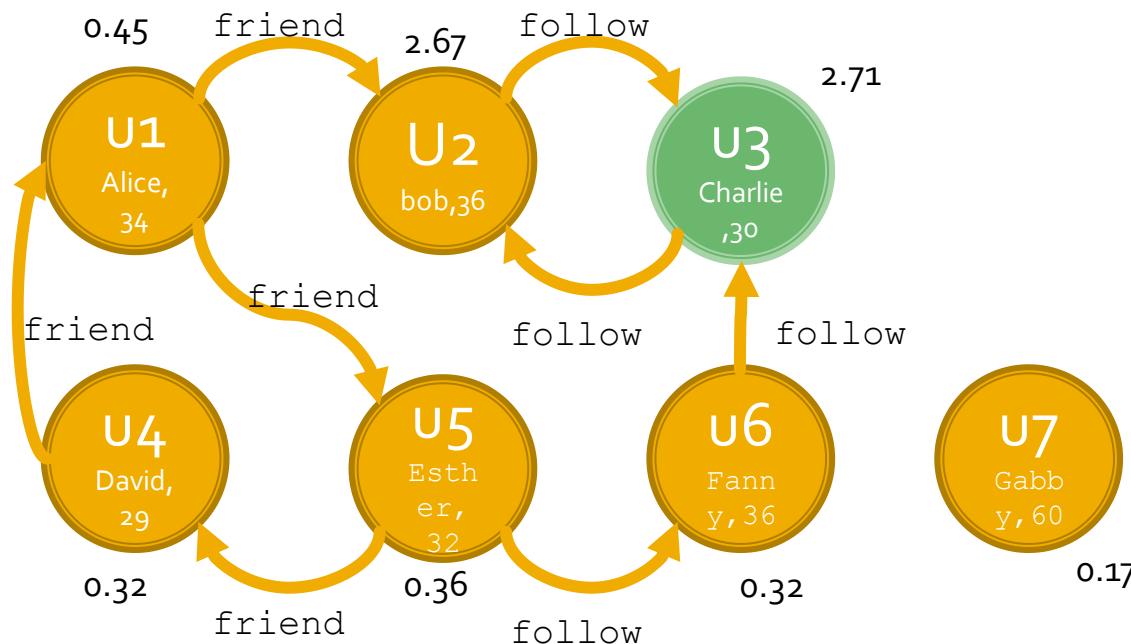
PageRank: Example

- Apply the PageRank algorithm on the following graph and select the user associated with the highest PageRank value



PageRank: Example

- Apply the PageRank algorithm on the following graph and select the user associated with the highest PageRank value



PageRank: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

PageRank: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

PageRank: Example

```
# Run the PageRank algorithm
pageRanks = g.pageRank(maxIter=30)

# Select the maximum value of PageRank
maxPageRank = pageRanks.vertices.agg({"pagerank":"max"})\
    .first()["max(pagerank)"]

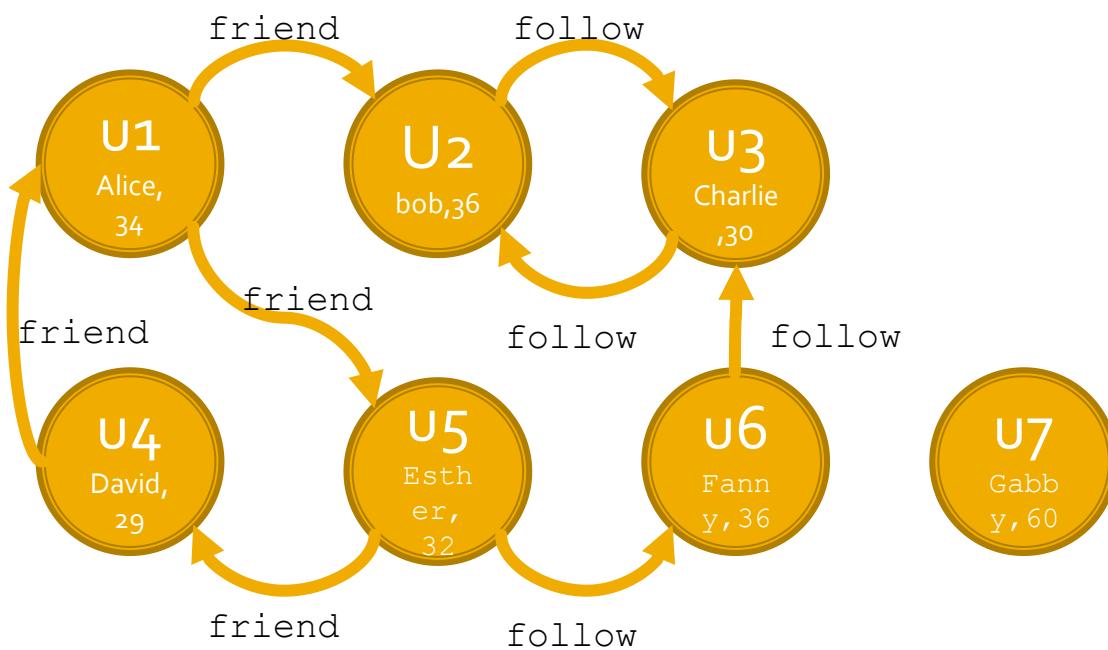
# Select the user with the maximum PageRank
pageRanks.vertices.filter(pageRanks.vertices.pagerank==maxPageRank) \
    .show()
```

Custom graph algorithms

- GraphFrames provides primitives for developing yourself other graph algorithms
- It is based on message passing approach
 - The two key components are:
 - aggregateMessages
 - Send messages between vertexes, and aggregate messages for each vertex
 - Joins
 - Join message aggregates with the original graph

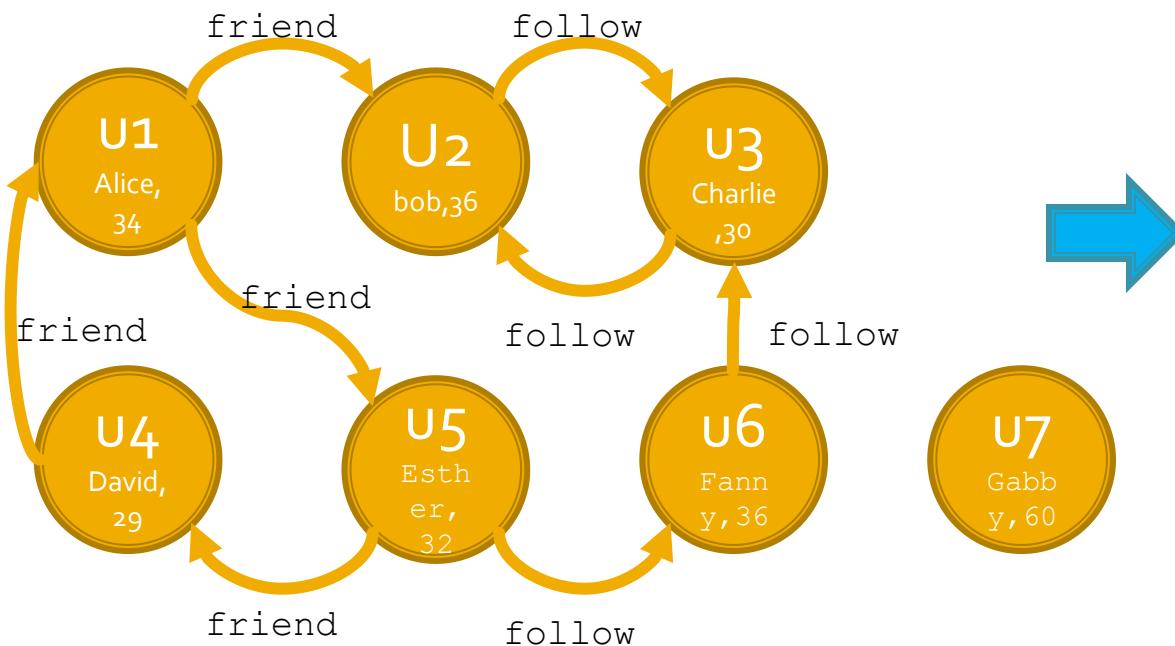
Custom graph algorithm: Example

- For each user, compute the sum of the ages of adjacent users (count many times the same adjacent user if there are many links)



Custom graph algorithm: Example

- For each user, compute the sum of the ages of adjacent users (count many times the same adjacent user if there are many links)



Vertex	Distance to u1
U1	97
U2	94
U3	108
U4	66
U5	99
U6	62

Custom graph algorithm: Example

```
from graphframes import GraphFrame
from pyspark.sql.functions import sum
from graphframes.lib import AggregateMessages

# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),\
                           ("u2", "Bob", 36),\
                           ("u3", "Charlie", 30),\
                           ("u4", "David", 29),\
                           ("u5", "Esther", 32),\
                           ("u6", "Fanny", 36),\
                           ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Custom graph algorithm: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                           ("u2", "u3", "follow"),\
                           ("u3", "u2", "follow"),\
                           ("u6", "u3", "follow"),\
                           ("u5", "u6", "follow"),\
                           ("u5", "u4", "friend"),\
                           ("u4", "u1", "friend"),\
                           ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])
```

```
# Create the graph
g = GraphFrame(v, e)
```

Custom graph algorithm: Example

```
# For each user, sum the ages of the adjacent users

# Send the age of each destination of an edge to its source
msgToSrc = AggregateMessages.dst["age"]

# Send the age of each source of an edge to its destination
msgToDst = AggregateMessages.src["age"]

# Aggregate messages
aggAge = g.aggregateMessages(sum(AggregateMessages.msg),\
    sendToSrc=msgToSrc,\n    sendToDst=msgToDst)

# Show result
aggAge.show()
```

Streaming data analytics: Frameworks

What is stream processing?

- Act of **continuously incorporating new data** to compute a result
- **Input data is unbounded** → no beginning and no end
- Series of events that arrive at the stream processing system
- The application will output multiple versions of the results as it runs or put them in a storage

Motivation

- Many important applications must process large streams of live data and provide results in **near-real-time**
 - Social network trends
 - Website statistics
 - Intrusion detection systems
 - ...

Advantages

- **Vastly higher throughput** in data processing
- **Low latency**: application respond quickly (e.g., in seconds)
 - It can keep states in memory
- **More efficient** in updating a result than repeated batch jobs, because it automatically incrementalizes the computation

Requirements and Challenges

- Scalable to large clusters
- Responding to events at low latency
- Simple programming model
- Processing each event exactly once despite machine failures - Efficient fault-tolerance in stateful computations

Requirements and Challenges

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Handling load imbalance and stragglers
- Updating your application's business logic at runtime

Stream Processing Frameworks for Big Streaming Data Analytics

- Several frameworks have been proposed to process in real-time or in near real-time data streams
 - Apache Spark (Streaming component)
 - Apache Storm
 - Apache Flink
 - Apache Samza
 - Apache Apex
 - Apache Flume
 - Amazon Kinesis Streams
 - ...
- All these frameworks use a cluster of servers to scale horizontally with respect to the (big) amount of data to be analyzed

Stream Processing Frameworks for Big Streaming Data Analytics

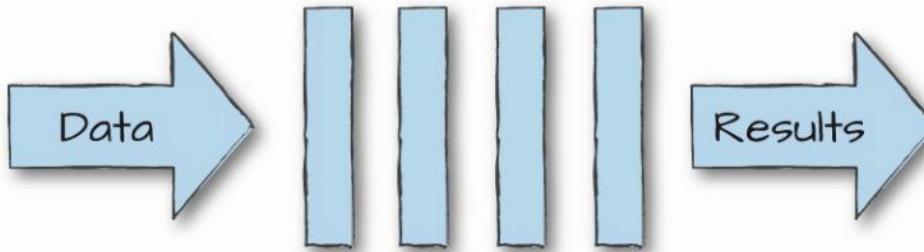
- Two main “solutions”
 - “Continuous” computation of data streams
 - Data are processed as soon as they arrive
 - Every time a new record arrives from the input stream, it is immediately processed and a result is emitted as soon as possible
 - Real-time processing

Stream Processing Frameworks for Big Streaming Data Analytics

- “Micro-batch” stream processing
 - Input data are collected in micro-batches
 - Each micro-batch contains all the data received in a time window (typically less than a few seconds of data)
 - One micro-batch a time is processed
 - Every time a micro-batch of data is ready, its entire content is processed and a result is emitted
 - Near real-time processing

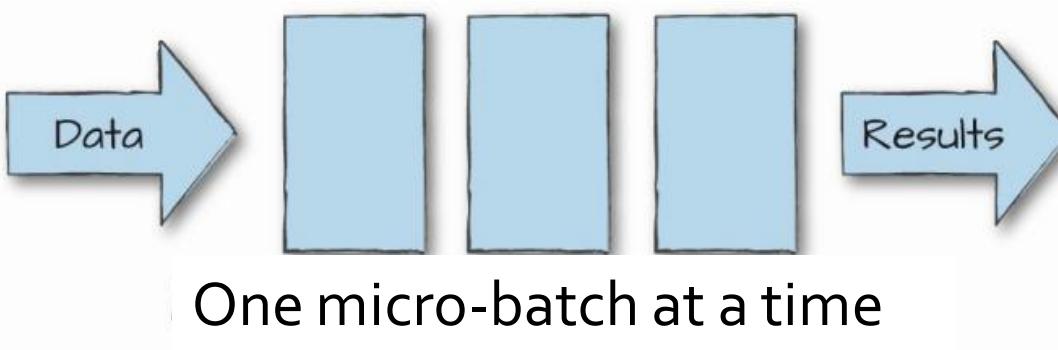
Continuous vs Micro-batch

- Continuous computation



One record at a time

- Micro-batch computation



One micro-batch at a time

Input data processing and Result guarantees

- **At-most-once**
 - Every input element of a stream is processed once or less
 - It is also called no guarantee
 - The result can be wrong/approximated
- **At-least-once**
 - Every input element of a stream is processed once or more
 - Input elements are replayed when there are failures
 - The result can be wrong/approximated

Input data processing and Result guarantees

- **Exactly-once**
 - Every input element of a stream is processed exactly once
 - Input elements are replayed when there are failures
 - If elements have been already processed they are not reprocessed
 - The result is always correct
 - Slower than the other processing approaches

Spark Streaming

What is Spark Streaming?

- Spark Streaming is a framework for large scale stream processing
 - Scales to 100s of nodes
 - Can achieve second scale latencies
 - Provides a simple batch-like API for implementing complex algorithm
 - **Micro-batch** streaming processing
 - **Exactly-once** guarantees
 - Can absorb live data streams from Kafka, Flume, ZeroMQ, Twitter, ...

What is Spark Streaming?



Motivation

- Many important applications must process large streams of live data and provide results in **near-real-time**
 - Social network trends
 - Website statistics
 - Intrusion detection systems
 - ...

Requirements

- Scalable to large clusters
- Second-scale latencies
- Simple programming model
- Efficient fault-tolerance in stateful computations

Spark Discretized Stream Processing

Discretized Stream Processing

- Spark streaming runs a streaming computation as a series of very small, deterministic batch jobs
- It splits each input stream in “portions” and processes one portion at a time (in the incoming order)
 - The same computation is applied on each portion of the stream
 - Each portion is called **batch**

Discretized Stream Processing

■ Spark streaming

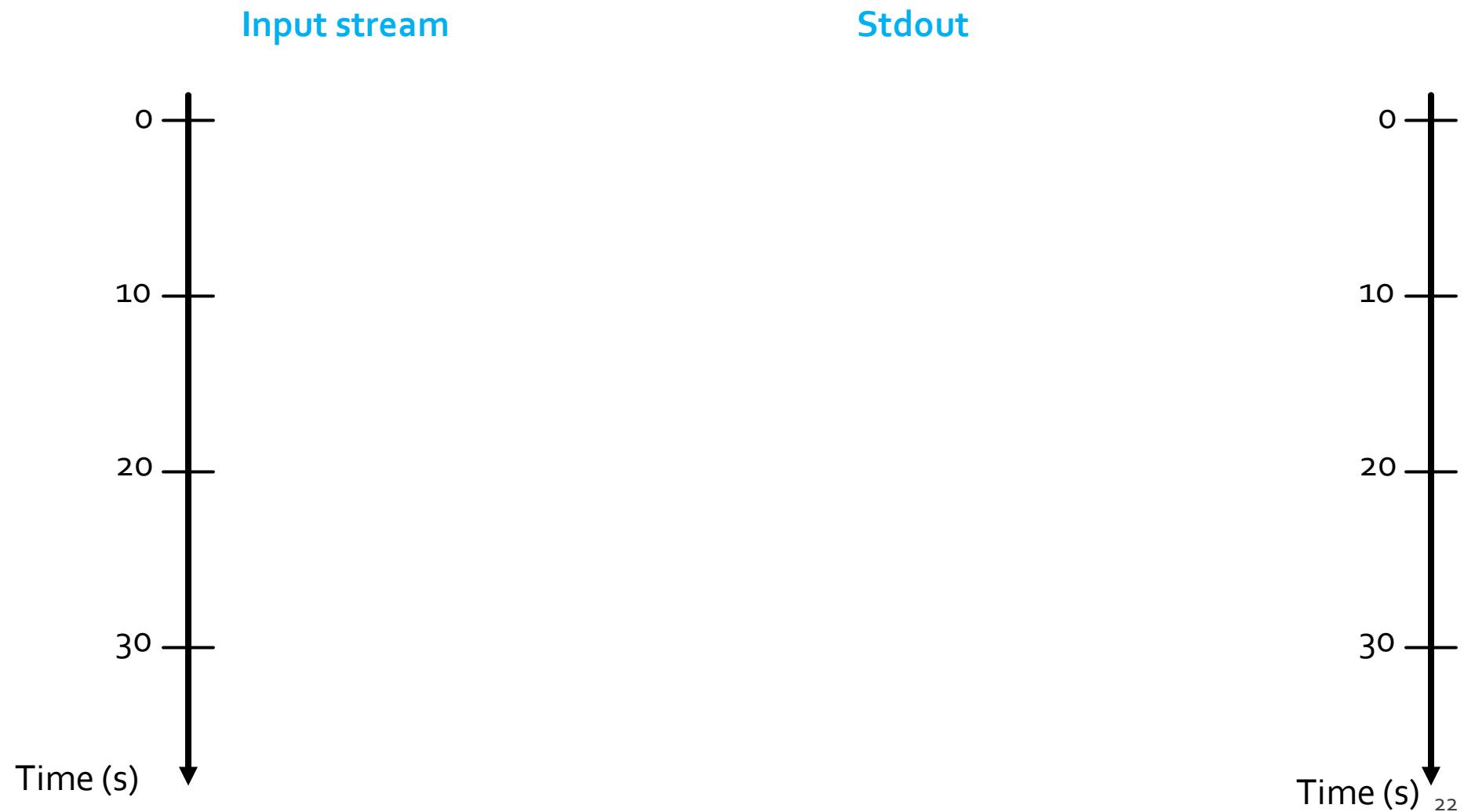
- Splits the live stream into batches of X seconds
- Treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



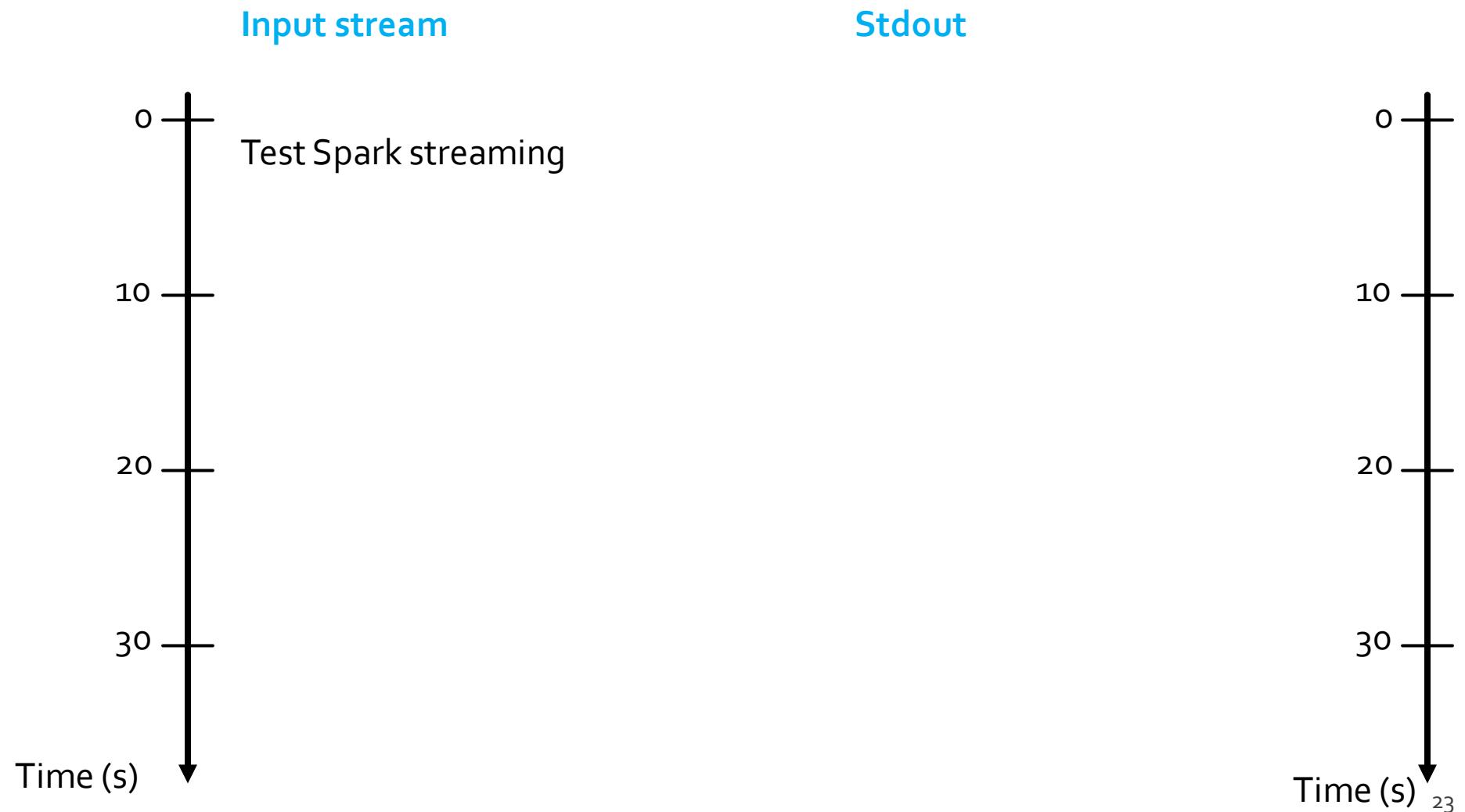
Word count – Spark Streaming version

- Problem specification
 - Input: a stream of sentences
 - Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
 - i.e., execute the word count application one time for each batch of 10 seconds

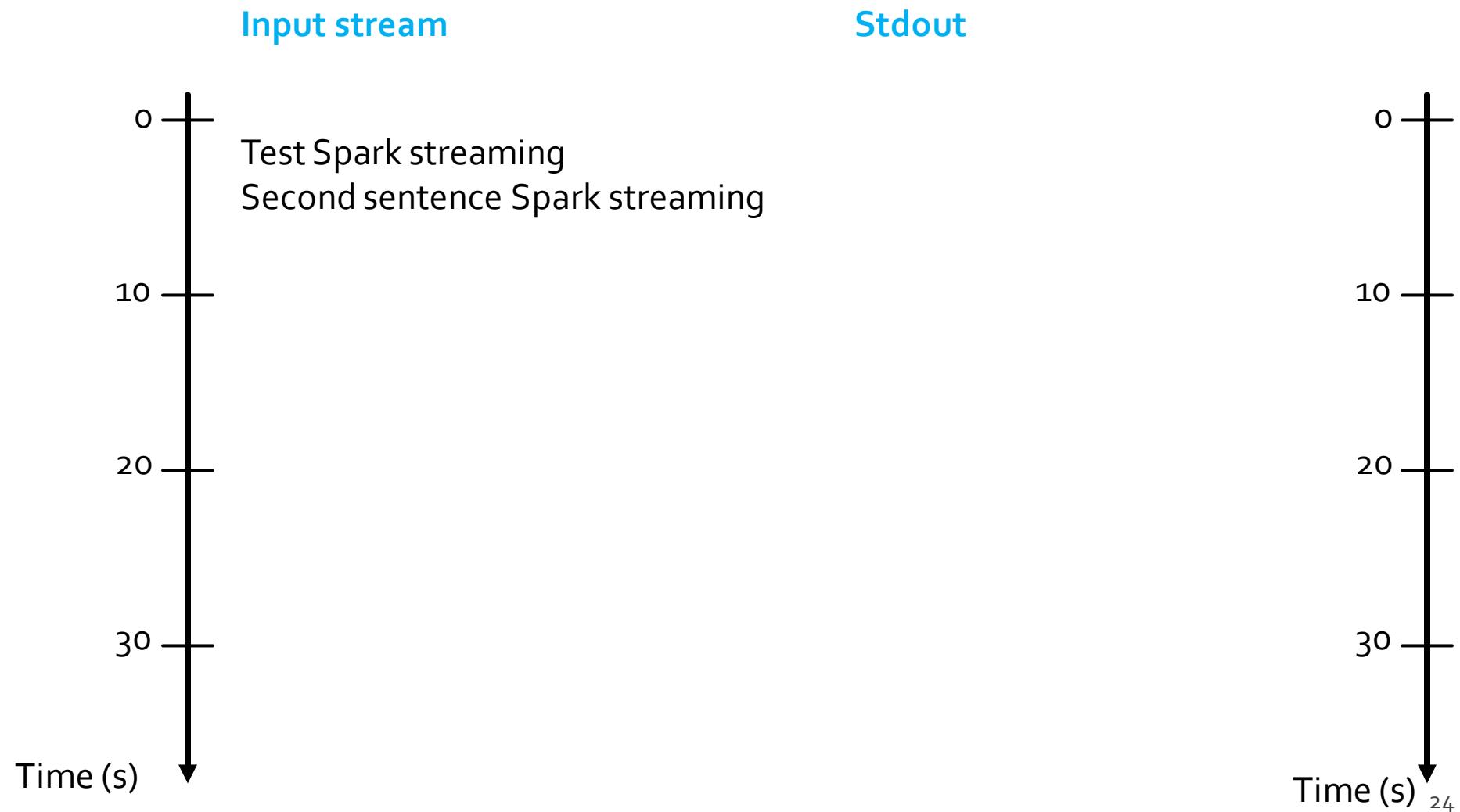
Word count – Spark Streaming version



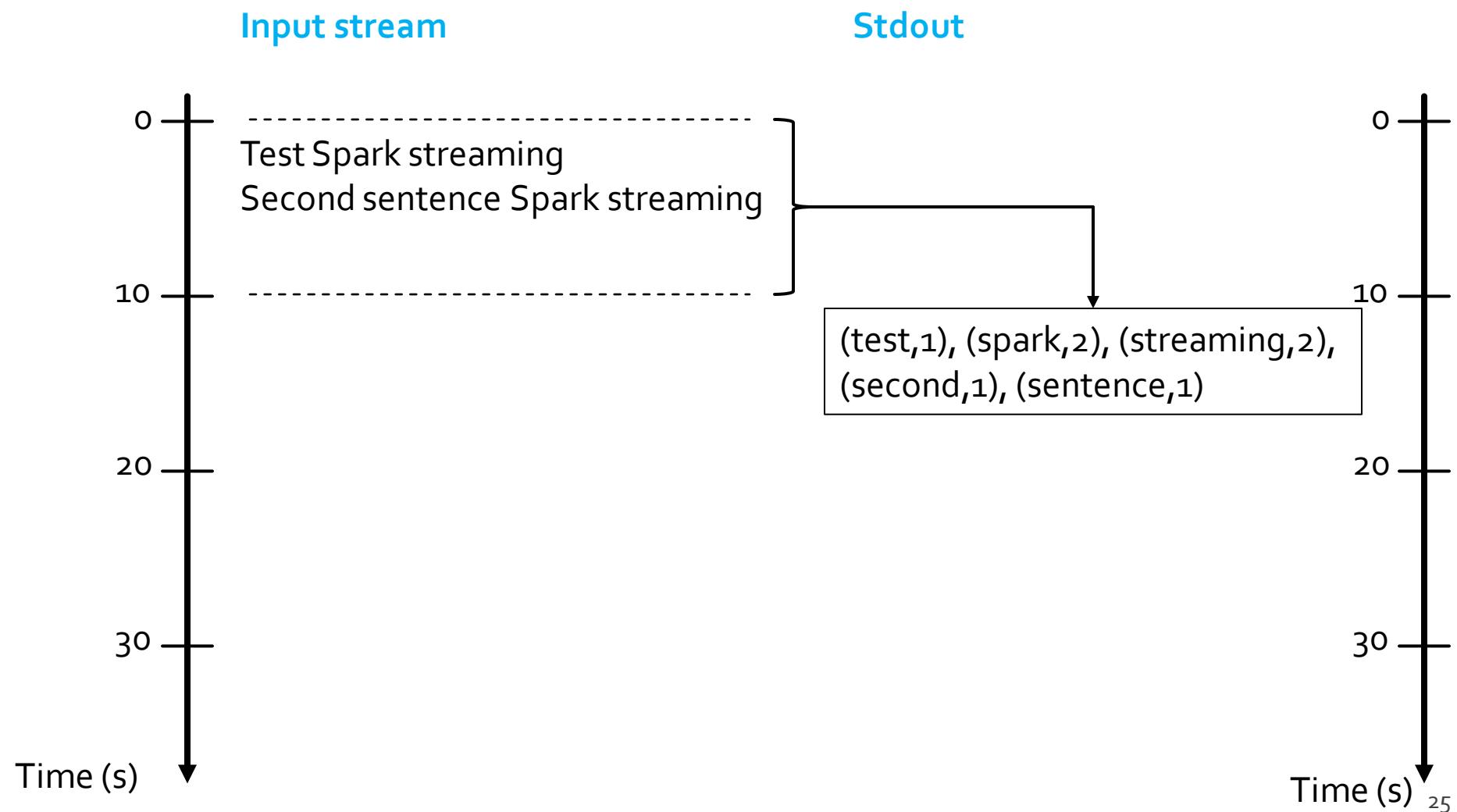
Word count – Spark Streaming version



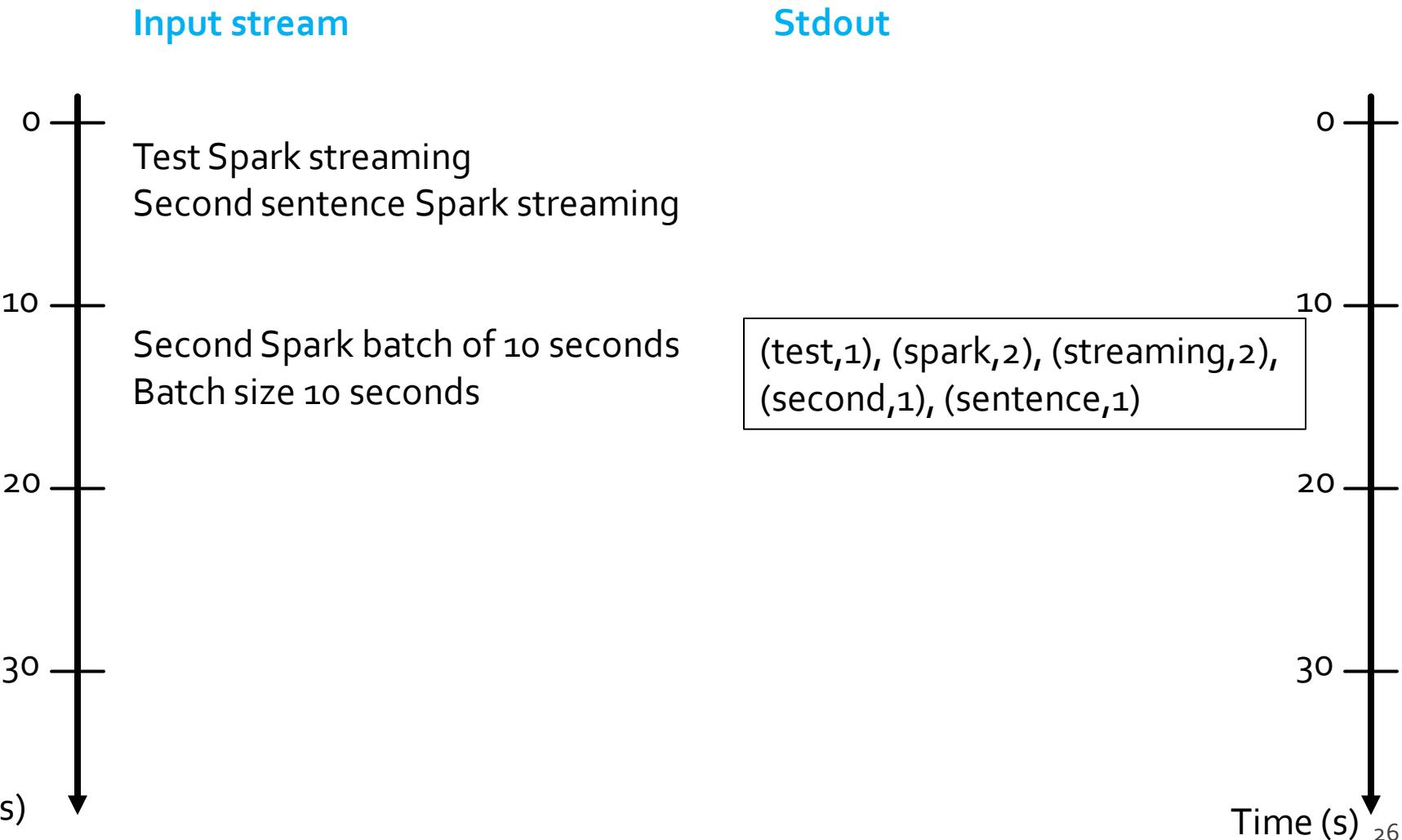
Word count – Spark Streaming version



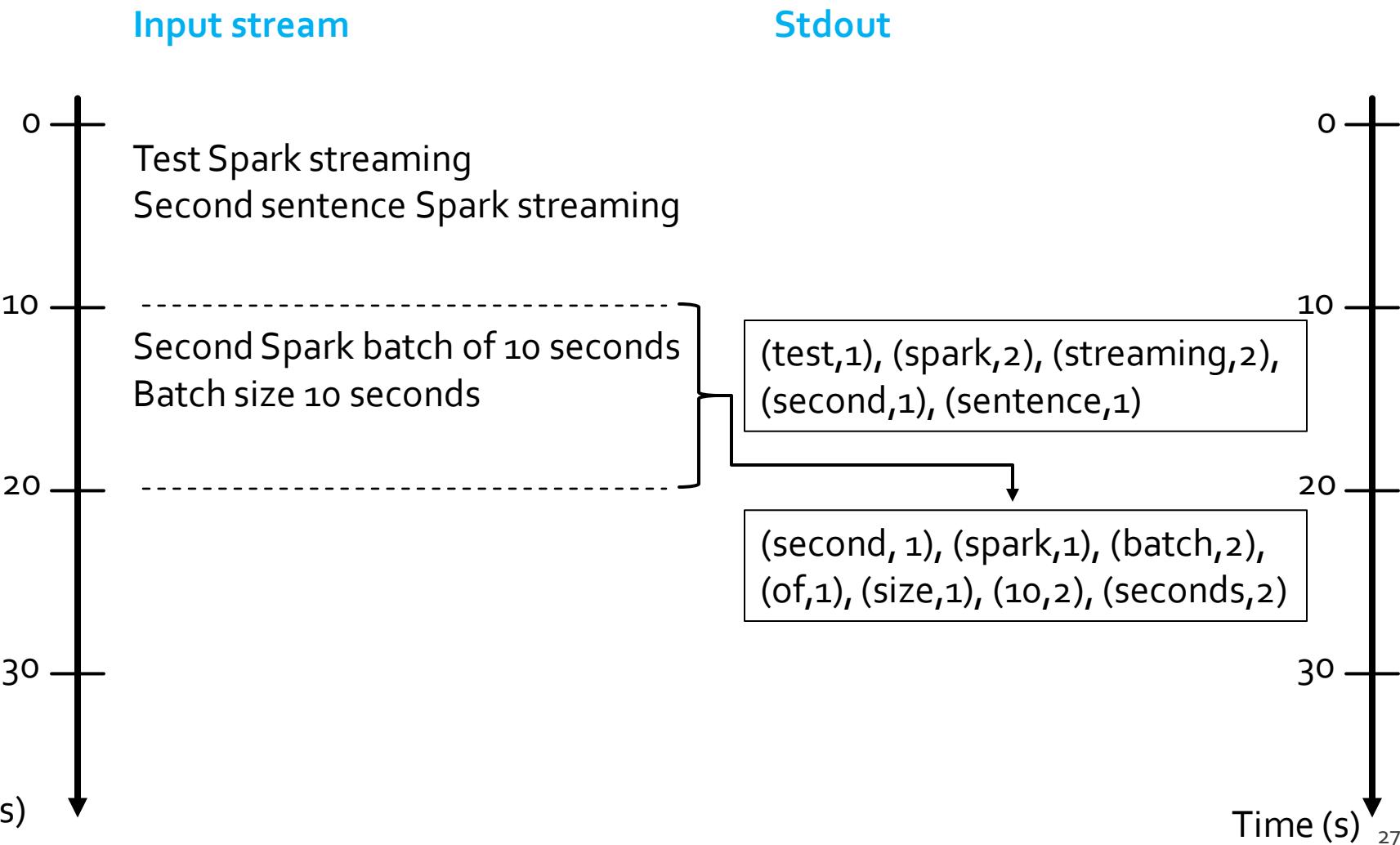
Word count – Spark Streaming version



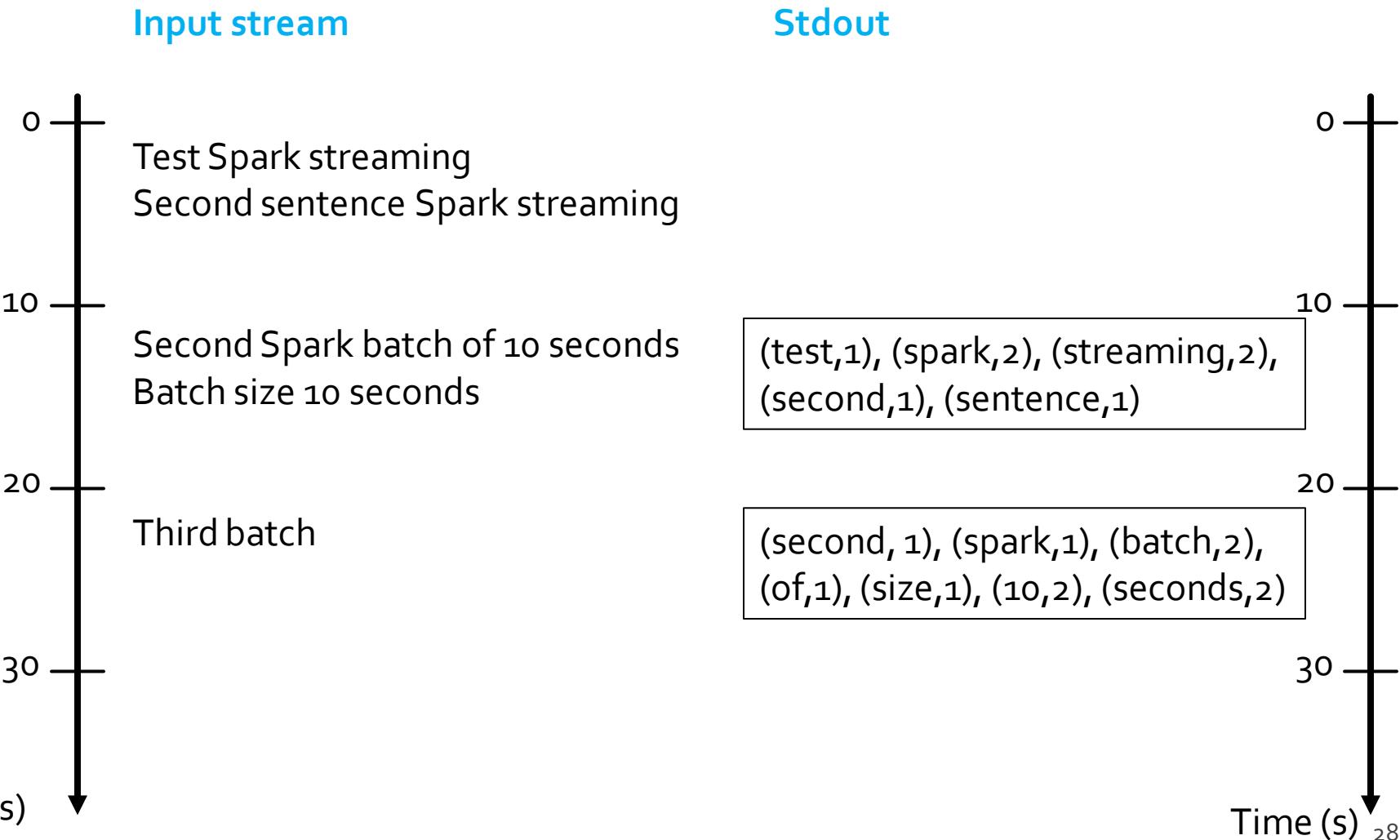
Word count – Spark Streaming version



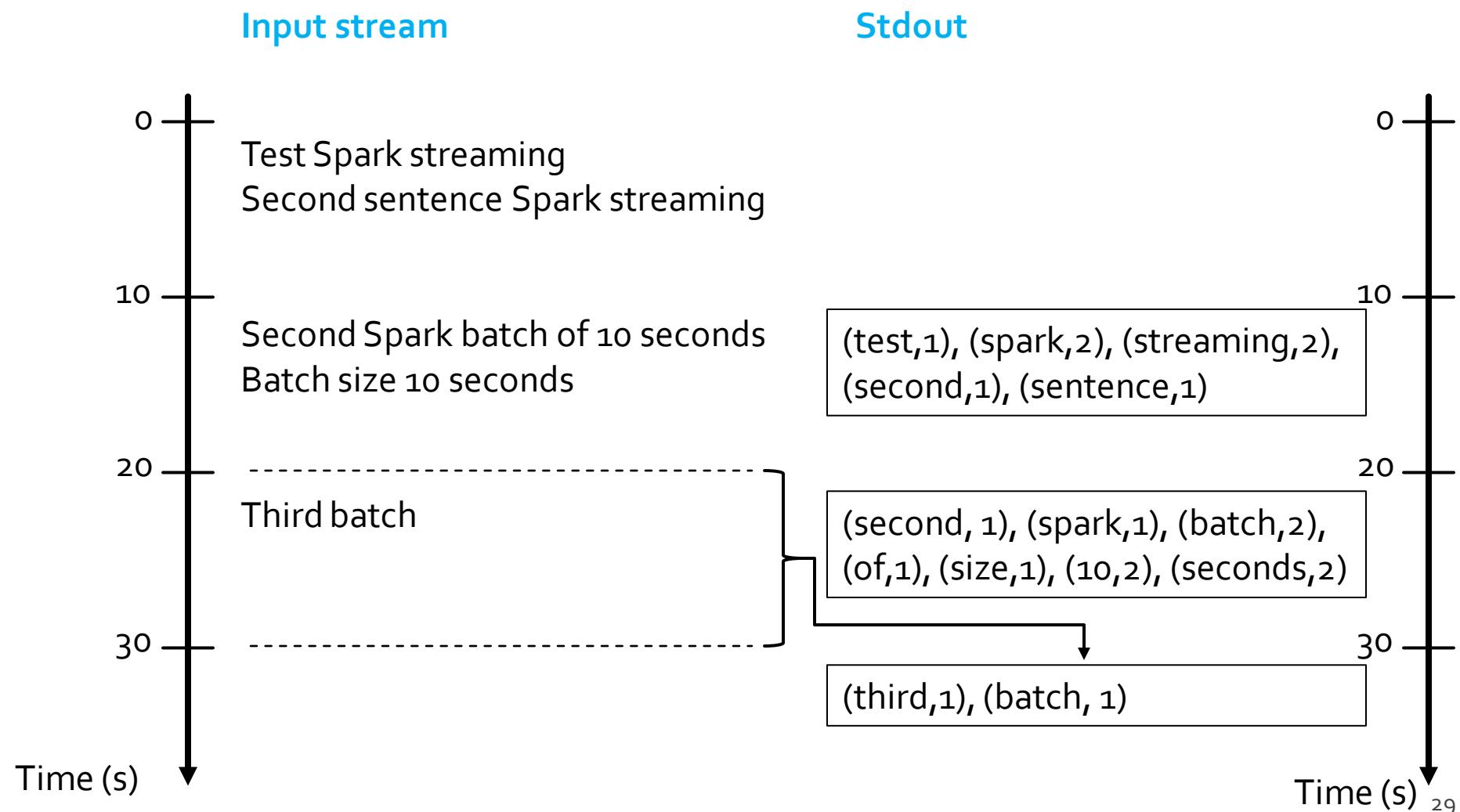
Word count – Spark Streaming version



Word count – Spark Streaming version



Word count – Spark Streaming version



Key concepts

- **DStream**
 - Sequence of RDDs representing a discretized version of the input stream of data
 - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets, ..
 - One RDD for each batch of the input stream

Key concepts

- **Transformations**

- Modify data from one DStream to another
- “Standard” RDD operations
 - map, countByValue, reduce, join, ...
- Window and Stateful operations
 - window, countByValueAndWindow, ...

- **Output Operations/Actions**

- Send data to external entity
 - saveAsHadoopFiles, saveAsTextFile, ...

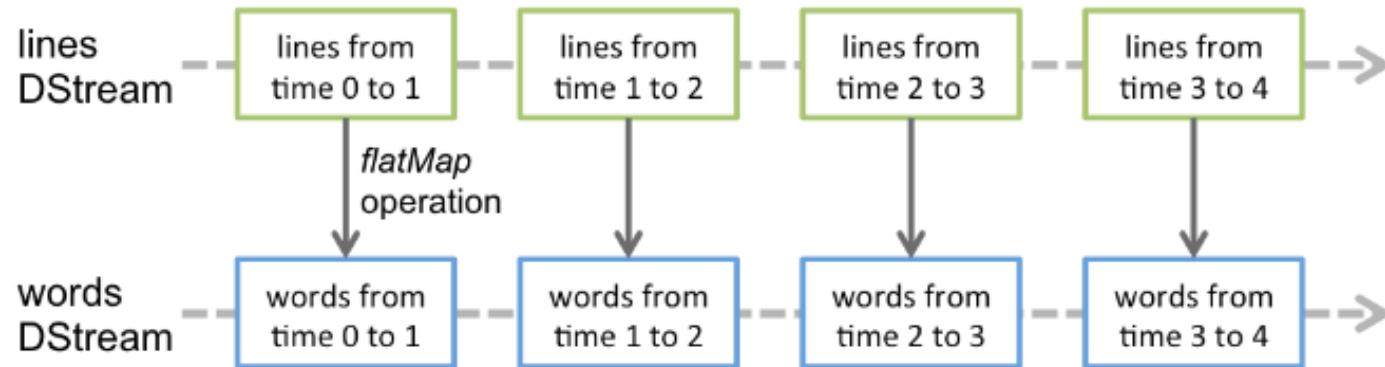
Word count – DStreams

- A DStream is represented by a continuous series of RDDs. Each RDD in a DStream contains data from a certain batch/interval



Word count – DStreams

- Any operation applied on a DStream translates to operations on the underlying RDDs
- These underlying RDD transformations are computed by the Spark engine



Fault-tolerance

- DStreams remember the sequence of operations that created them from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data

Spark Streaming Programs

Basic Structure of a Spark Streaming Program (1)

- Define a Spark Streaming Context object
 - Define the size of the batches (in seconds) associated with the Streaming context
- Specify the input stream and define a DStream based on it
- Specify the operations to execute for each batch of data
 - Use transformations and actions similar to the ones available for “standard” RDDs

Basic Structure of a Spark Streaming Program (2)

- Invoke the start method
 - To start processing the input stream
- Wait until the application is killed or the timeout specified in the application expires
 - If the timeout is not set and the application is not killed **the application will run forever**

Spark Streaming Context

- The Spark Streaming Context is defined by using the `StreamingContext(SparkConf sparkC, Duration batchDuration)` constructor of the class `pyspark.streaming.StreamingContext`
- The `batchDuration` parameter specifies the “size” of the batches in seconds
- Example
 - from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 10)
 - The input streams associated with this context will be split in batches of 10 seconds

Spark Streaming Context

- After a context is defined, you have to do the following
 - Define the input sources by creating input Dstreams
 - Define the streaming computations by applying transformation and output operations to DStreams

Input Streams

- The input Streams can be generate from different sources
 - TCP socket, Kafka, Flume, Kinesis, Twitter
 - Also an HDFS folder can be used as “input stream”
 - This option is usually used during the application development to perform a set of initial tests

Input Streams: TPC socket

- A DStream can be associated with the content emitted by a TCP socket
- `socketTextStream(String hostname, int port_number)` is used to create a DStream based on the textual content emitted by a TPC socket
- Example
`lines = ssc.socketTextStream("localhost", 9999)`
 - “Store” the content emitted by localhost:9999 in the lines DStream

Input Streams: (HDFS) folder

- A DStream can be associated with the content of an input (HDFS) folder
 - Every time a **new file** is inserted in the folder, the content of the file is “stored” in the associated DStream and processed
 - Pay attention that updating the content of a file does not trigger/change the content of the DStream
- **textFileStream(String folder)** is used to create a DStream based on the content of the input folder

Input Streams: (HDFS) folder

- Example

```
lines = textFileStream(inputFolder)
```

- “Store” the content of the files inserted in the input folder in the lines Dstream
- Every time new files are inserted in the folder their content is “stored” in the current “batch” of the stream

Input Streams: other sources

- Usually DStream objects are defined on top of streams emitted by specific applications that emit real-time streaming data
 - E.g., Apache Kafka, Apache Flume, Kinesis, Twitter
- You can also write your own applications for generating streams of data
 - However, Kafka, Flume and similar tools are usually a more reliable and effective solutions for generating streaming data

Transformations

- Analogously to standard RDDs, also DStreams are characterized by a set of transformations
 - When applied to DStream objects, transformations return a new DStream Object
 - The transformation is applied on one batch (RDD) of the input DStream at a time and returns a batch (RDD) of the new DStream
 - i.e., each batch (RDD) of the input DStream is associated with exactly one batch (RDD) of the returned DStream
- Many of the available transformations are the same transformations available for standard RDDs

Basic Transformations on DStreams

- **map(func)**
 - Returns a new DStream by passing each element of the source DStream through a function **func**
- **flatMap(func)**
 - Each input item can be mapped to 0 or more output items. Returns a new DStream
- **filter(func)**
 - Returns a new DStream by selecting only the records of the source DStream on which **func** returns true

Basic Transformations on DStreams

■ **reduce(func)**

- Returns a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function **func**
 - The function must be associative and commutative so that it can be computed in parallel
- Note that the **reduce** method of DStreams is a **transformation**

Basic Transformations on DStreams

- **reduceByKey(func)**
 - When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function
- **combineByKey(createCombiner, mergeValue, mergeCombiners)**
 - When called on a DStream of (K, V) pairs, returns a new DStream of (K, W) pairs where the values for each key are aggregated using the given combine functions

Basic Transformations on DStreams

■ **groupByKey()**

- When called on a DStream of (K, V) pairs, returns a new DStream of (K, Iterable<V>) pairs where the values for each key is the “concatenation” of all the values associated with key K
 - I.e., It returns a new DStream by applying groupByKey on one batch (one RDD) of the input stream at a time

Basic Transformations on DStreams

- **countByValue()**

- When called on a DStream of elements of type K, returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each batch of the source Dstream
- Note that the **countByValue** method of DStreams is a **transformation**

Basic Transformations on DStreams

■ **count()**

- Returns a new DStream of single-element RDDs by counting the number of elements in each batch (RDD) of the source Dstream
 - i.e., it counts the number of elements in each input batch (RDD)
- Note that the **count** method of DStreams is a **transformation**

Basic Transformations on DStreams

- **union(otherStream)**

- Returns a new DStream that contains the union of the elements in the source DStream and otherDStream

- **join(otherStream)**

- When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of $(K, (V, W))$ pairs with all pairs of elements for each key

Basic Transformations on DStreams

- **cogroup(otherStream)**
 - When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of $(K, Seq[V], Seq[W])$ tuples

Basic Actions on DStreams

- **pprint()**
 - Prints the first 10 elements of every batch of data in a DStream on the standard output of the **driver node** running the streaming application
 - Useful for development and debugging

Basic Actions on DStreams

- **saveAsTextFiles(prefix, [suffix])**
 - Saves the content of the DStream on which it is invoked as text files
 - One folder for each batch
 - The folder name at each batch interval is generated based on prefix, time of the batch (and suffix): "prefix-TIME_IN_MS[.suffix]"
 - Example

```
Counts.saveAsTextFiles(outputPathPrefix, "")
```

Start and run the computation

- The `streamingContext.start()` method is used to start the application on the input stream(s)
- The `awaitTerminationOrTimeout(long millisecs)` method is used to specify how long the application will run
- The `awaitTermination()` method is used to **run** the application **forever**
 - Until the application is explicitly killed
 - The processing can be manually stopped using `streamingContext.stop()`

Start and run the computation

- Points to remember:

- Once a context has been started, no new streaming computations can be set up or added to it
- Once a context has been stopped, it cannot be restarted
- Only one StreamingContext per application can be active at the same time
- **stop()** on StreamingContext also stops the SparkContext
 - To stop only the StreamingContext, set the optional parameter of stop() called **stopSparkContext** to False

Example: Word count – Spark Streaming version

- Problem specification
 - Input: a stream of sentences retrieved from localhost:9999
 - Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
 - i.e., execute the word count problem for each batch of 5 seconds
 - Store the results also in an HDFS folder

Example: Word count – Spark Streaming version

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create a configuration object and#set the name of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create a Spark Context object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

Example: Word count – Spark Streaming version

```
# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)

# Print the result on the standard output
wordsCounts.pprint()

# Store the result in HDFS
wordsCounts.saveAsTextFiles(outputPathPrefix, "")
```

Example: Word count – Spark Streaming version

```
#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

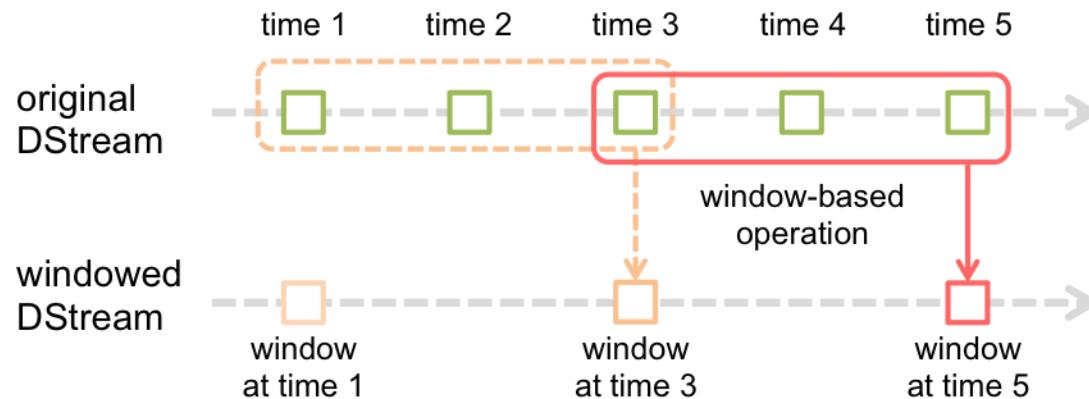
Windowed Computation

Window operation

- Spark Streaming also provides windowed computations
 - It allows you to apply transformations over a sliding window of data
 - Each window contains a set of batches of the input stream
 - Windows can be overlapped
 - i.e., the same batch can be included in many consecutive windows

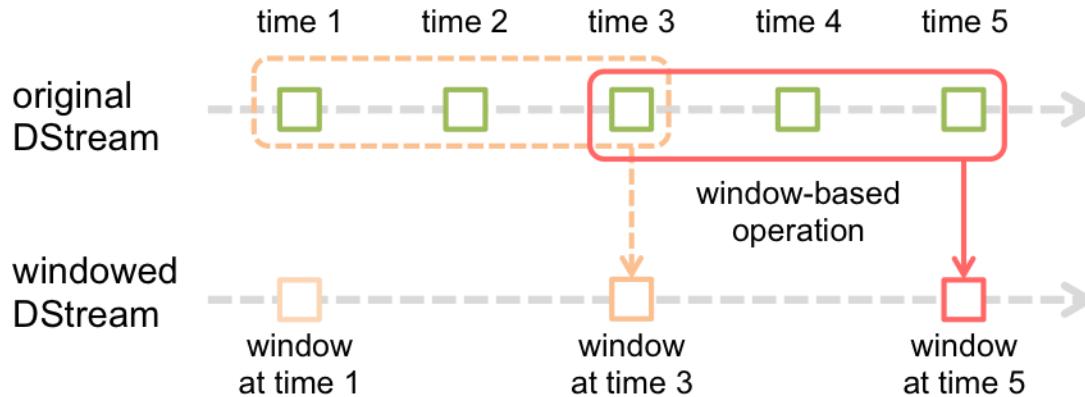
Window operation

- Graphical example



- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream

Window operation



- In the graphical example, the operation
 - is applied over the last 3 time units of data (i.e., the last 3 batches of the input DStream)
 - Each window contains the data of 3 batches
 - slides by 2 time units

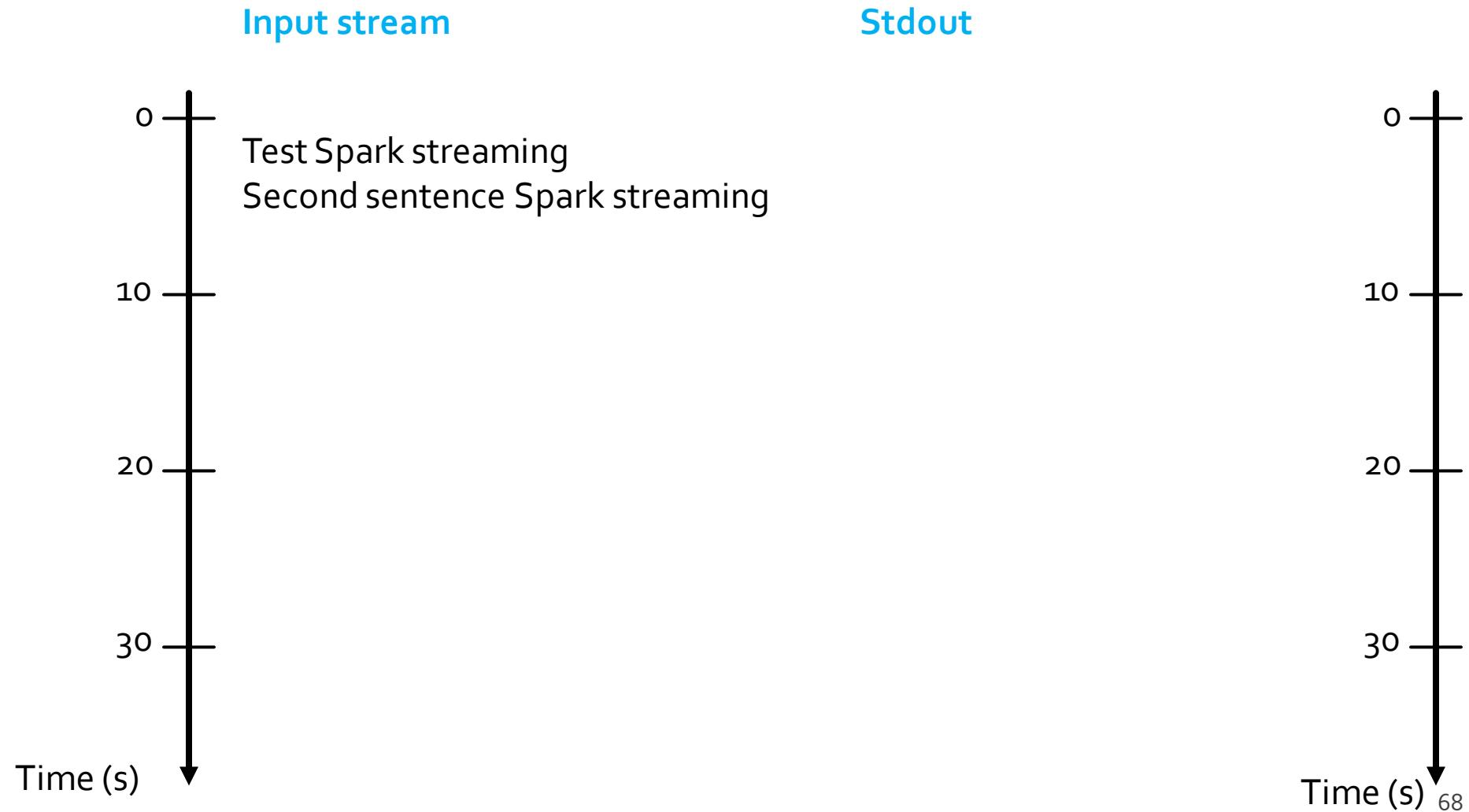
Window operation: parameters

- Any window operation needs to specify two parameters:
 - Window length
 - The duration of the window (3 in the example)
 - Sliding interval
 - The interval at which the window operation is performed (2 in the example)
- These two parameters must be multiples of the batch interval of the source DStream

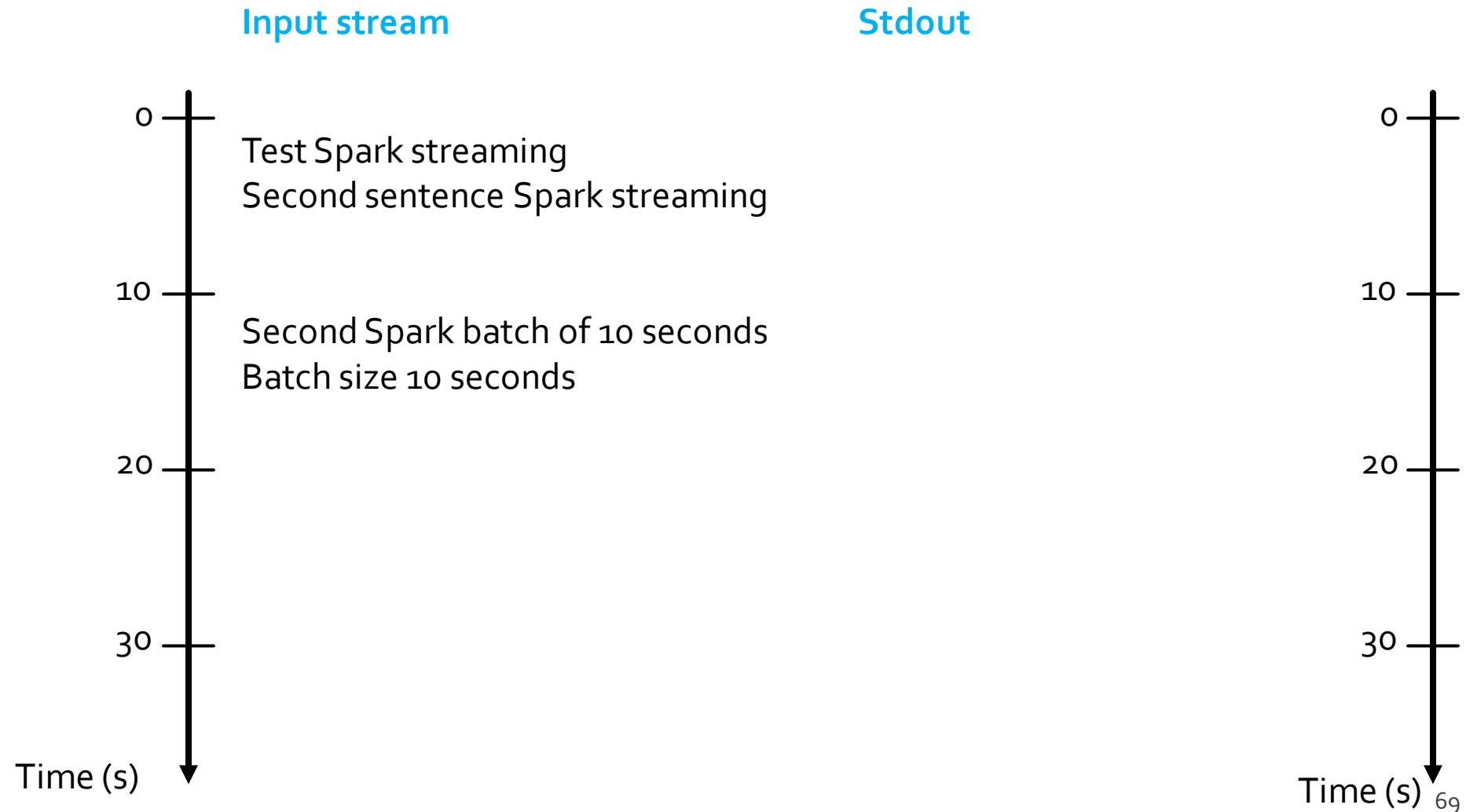
Word count and Window

- Problem specification
 - Input: a stream of sentences
 - Split the input stream in batches of 10 seconds
 - Define windows with the following characteristics
 - Window length: 20 seconds (i.e., 2 batches)
 - Sliding interval: 10 seconds (i.e., 1 batch)
 - Print on the standard output, for each window, the occurrences of each word appearing in the window
 - i.e., execute the word count problem for each window

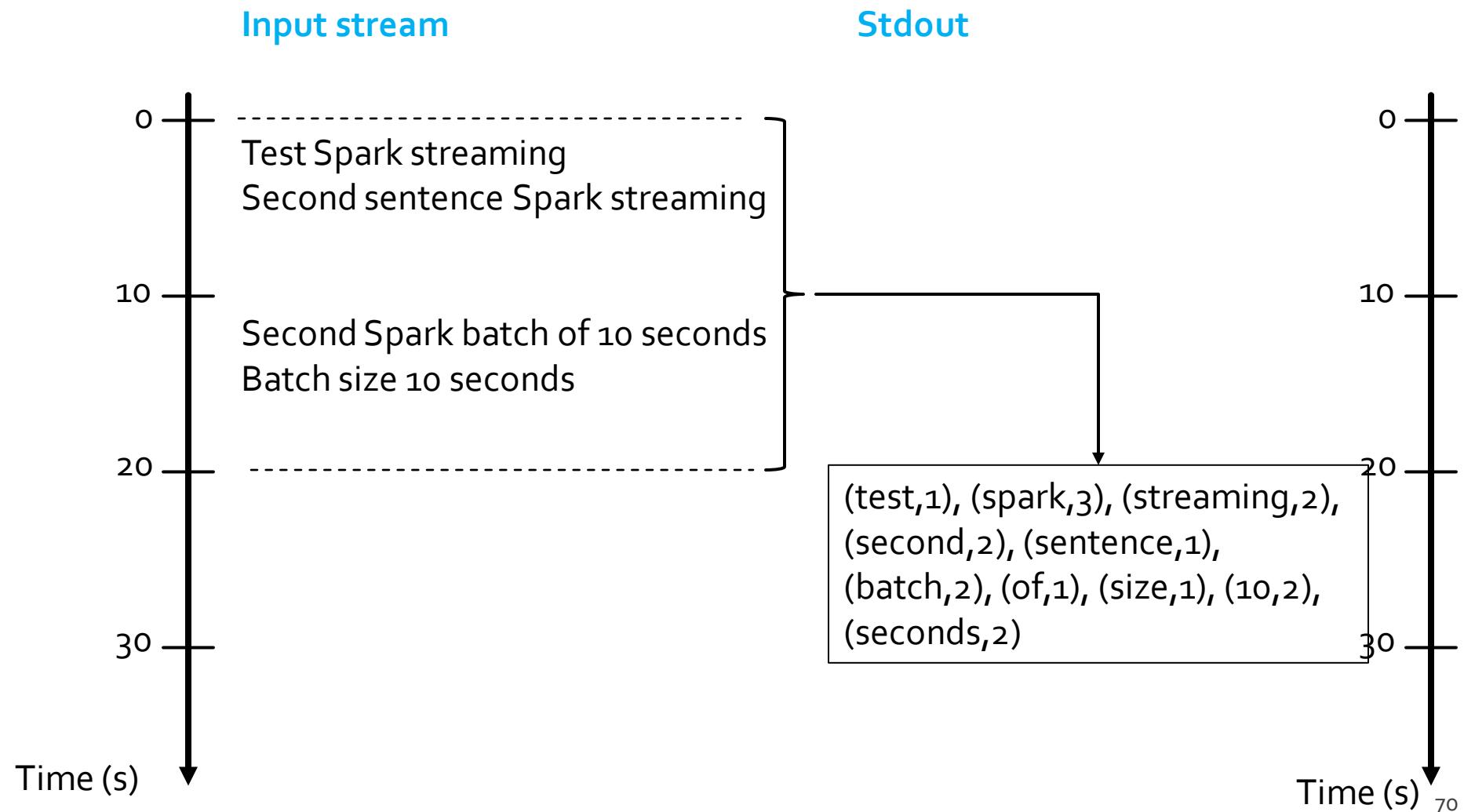
Word count and Window



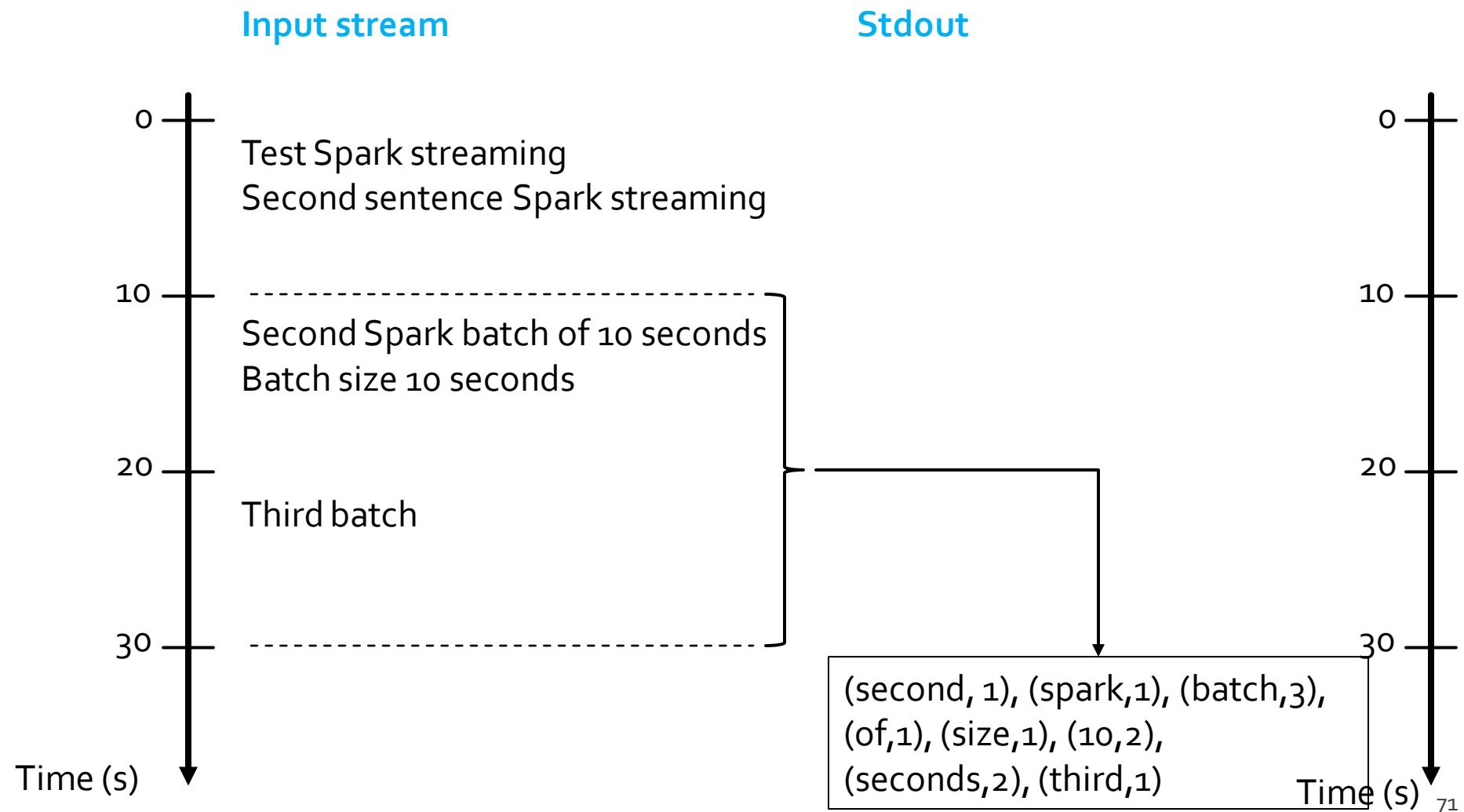
Word count and Window



Word count and Window



Word count and Window



Basic Window Transformations

- **window(windowLength, slideInterval)**
 - Returns a new DStream which is computed based on windowed batches of the source DStream
- **countByWindow(windowLength, slideInterval)**
 - Returns a new single-element stream containing the number of elements of each window
 - The returned object is a Dstream of Long objects. However, it contains only one value for each window (the number of elements of the last analyzed window)

Basic Window Transformations

- **reduceByWindow(reduceFunc, invReduceFunc, windowDuration, slideDuration)**
 - Returns a new single-element stream, created by aggregating elements in the stream over a sliding interval using **func**
 - The function must be associative and commutative so that it can be computed correctly in parallel
 - If **invReduceFunc** is not None, the reduction is done incrementally using the old window's reduced value

Basic Window Transformations

- **countByValueAndWindow(windowDuration , slideDuration)**
 - When called on a DStream of elements of type K, returns a new DStream of (K, Long) pairs where the value of each key K is its frequency in each window of the source DStream

Basic Window Transformations

- `reduceByKeyAndWindow(func, invFunc, windowDuration, slideDuration=None, numPartitions=None)`
 - When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function **func over batches in a sliding window**
 - The window duration (length) is specified as a parameter of this invocation (**windowDuration**)

Basic Window Transformations

- If `slideDuration` is `None`, the `batchDuration` of the `StreamingContext` object is used
 - i.e., 1 batch sliding window
- If `invFunc` is provided (is not `None`), the reduction is done incrementally using the old window's reduced values
 - i.e., `invFunc` is used to apply an inverse reduce operation by considering the old values that left the window (e.g., subtracting old counts)

Checkpoints

- A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)
- For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures
- This result is achieved by means of checkpoints
 - Operations that store the data and metadata needed to restart the computation if failures happen
- Checkpointing is necessary even for some window transformations and stateful transformations

Checkpoints

- Checkpointing is enabled by using the **checkpoint(String folder)** method of SparkStreamingContext
 - The parameter is the folder that is used to store temporary data
- Similar as for processing graphs with GraphFrames library
 - With GraphFrames, the checkpoint was the one of SparkContext

Example: Word count and Windows

■ Problem specification

- Input: a stream of sentences retrieved from localhost:9999
- Split the input stream in batches of 5 seconds
- Define windows with the following characteristics
 - Window length: 15 seconds (i.e., 3 batches)
 - Sliding interval: 5 seconds (i.e., 1 batch)
- Print on the standard output, for each window, the occurrences of each word appearing in the window
 - i.e., execute the word count problem for each window
- Store the results also in an HDFS folder

Example: Word count and Windows

```
from pyspark.streaming import StreamingContext  
  
# Set prefix of the output folders  
outputPathPrefix="resSparkStreamingExamples"  
  
#Create a configuration object and set the name of the applicationconf  
SparkConf().setAppName("Streaming word count")  
  
# Create a Spark Context object  
sc = SparkContext(conf=conf)  
  
# Create a Spark Streaming Context object  
ssc = StreamingContext(sc, 5)  
  
# Set the checkpoint folder (it is needed by some window transformations)  
ssc.checkpoint("checkpointfolder")
```

Example: Word count and Windows

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

# reduceByKeyAndWindow is used instead of reduceByKey
# The duration of the window is also specified
wordsCounts = wordsOnes\
.reduceByKeyAndWindow(lambda v1, v2: v1+v2, None, 15)
```

Example: Word count and Windows

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
wordsCounts.pprint()

# Store the output of the computation in the folders with prefix
# outputPathPrefix
wordsCounts.saveAsTextFiles(outputPathPrefix, "")

#Start the computation
ssc.start()

ssc.awaitTermination()
```

Example: Word count and Windows – V2

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create a configuration object and#set the name of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create a Spark Context object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Set the checkpoint folder (it is needed by some window transformations)
ssc.checkpoint("checkpointfolder");
```

Example: Word count and Windows – V2

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

# reduceByKeyAndWindow is used instead of reduceByKey
# The duration of the window is also specified
wordsCounts = wordsOnes\
.reduceByKeyAndWindow(lambda v1, v2: v1+v2, lambda vnow, vold: vnow-vold, 15)
```

Example: Word count and Windows – V2

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

# reduceByKeyAndWindow is used instead of reduceByKey
# The duration of the window is also specified
wordsCounts = wordsOnes\
    .reduceByKeyAndWindow(lambda v1, v2: v1+v2, lambda vnow, vold: vnow-vold, 15)
```

In this solution the inverse function is also specified
in order to compute the result incrementally

Example: Word count and Windows – V2

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
wordsCounts.pprint()

# Store the output of the computation in the folders with prefix
# outputPathPrefix
wordsCounts.saveAsTextFiles(outputPathPrefix, "")

#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

Stateful Computation

UpdateStateByKey Transformation

- The `updateStateByKey` transformation allows maintaining a “state” for each key
 - The value of the state of each key is continuously updated every time a new batch is analyzed

UpdateStateByKey Transformation

- The use of updateStateByKey is based on two steps
 - Define the state
 - The data type of the state associated with the keys can be an arbitrary data type
 - Define the state update function
 - Specify with a function how to update the state of a key using the previous state and the new values from an input stream associated with that key

UpdateStateByKey Transformation

- In every batch, Spark will apply the state update function for all existing keys
- For each key, the update function is used to update the value associated with a key by combining the former value and the new values associated with that key
 - For each key, the call method of the “function” is invoked on the list of new values and the former state value and returns the new aggregated value for the considered key

Word count and UpdateStateByKey Transformation

- By using the `UpdateStateByKey`, the application can continuously update the number of occurrences of each word
 - The number of occurrences stored in the DStream returned by this transformation is computed over the union of all the batches (from the first one to the current one)
 - For efficiency reasons, the new value for each key is computed by combining the last value for that key with the values of the current batch for the same key

Example: Word count - stateful version

- Problem specification
 - Input: a stream of sentences retrieved from localhost:9999
 - Split the input stream in batches of 5 seconds
 - Print on the standard output, every 5 seconds, the occurrences of each word appearing in the stream (from time 0 to the current time)
 - i.e., execute the word count problem from the beginning of the stream to current time
 - Store the results also in an HDFS folder

Example: Word count - stateful version

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create a configuration object and set the name of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create a Spark Context object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Set the checkpoint folder (it is needed by some window transformations)
ssc.checkpoint("checkpointfolder")
```

Example: Word count - stateful version

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)

# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
```

```
def updateFunction(newValues, currentCount):
```

```
    if currentCount is None:
```

```
        currentCount = 0
```

```
# Sum the new values to the previous state for the current key
```

```
return sum(newValues, currentCount)
```

This function is invoked one time for each key

```
# DStream made of cumulative counts for each key that get updated in every batch
```

```
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time  
def updateFunction(newValues, currentCount):
```

```
    if currentCount is None:
```

```
        currentCount = 0
```

Current state/value for the current key

```
# Sum the new values to the previous state for the current key  
return sum(newValues, currentCount)
```

```
# DStream made of cumulative counts for each key that get updated in every batch  
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time  
def updateFunction(newValues, currentCount):
```

```
    if currentCount is None:
```

```
        currentCount = 0
```

List of new integer values for the current key

```
    # Sum the new values to the previous state for the current key  
    return sum(newValues, currentCount)
```

```
# DStream made of cumulative counts for each key that get updated in every batch  
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)

# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Combine current state and new values

Example: Word count - stateful version

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
totalWordsCounts.pprint()

# Store the output of the computation in the folders with prefix
# outputPathPrefix
totalWordsCounts.saveAsTextFiles(outputPathPrefix, "")

#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

Transform transformation

Transform transformation

- Some types of transformations are not available for DStreams
 - E.g., sortBy, sortByKey, distinct()
- Moreover, sometimes you need to combine DStreams and RDDs
 - For example, the functionality of joining every batch in a data stream with another dataset (a “standard” RDD) is not directly exposed in the DStream API
- The **transform()** transformation can be used in these situations

Transform transformation

- **transform(func)**
 - It is a specific transformation of DStreams
 - It returns a new DStream by applying an RDD-to-RDD function to every RDD of the source Dstream
 - This can be used to apply arbitrary RDD operations on the DStream

Example: Word count – Use of transform

- Problem specification
 - Input: a stream of sentences retrieved from localhost:9999
 - Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
 - **The pairs must be returned/displayed sorted by decreasing number of occurrences (per batch)**
 - Store the results also in an HDFS folder

Example: Word count – Use of transform

```
from pyspark.streaming import StreamingContext  
  
# Set prefix of the output folders  
outputPathPrefix="resSparkStreamingExamples"  
  
#Create a configuration object and set the name of the applicationconf  
SparkConf().setAppName("Streaming word count")  
  
# Create a Spark Context object  
sc = SparkContext(conf=conf)  
  
# Create a Spark Streaming Context object  
ssc = StreamingContext(sc, 5)  
  
# Create a (Receiver) DStream that will connect to localhost:9999  
lines = ssc.socketTextStream("localhost", 9999)
```

Example: Word count – Use of transform

```
# Apply a chain of transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)

# Sort the content/the pairs by decreasing value (# of occurrences)
wordsCountsSortByKey = wordsCounts\
.transform(lambda batchRDD: batchRDD.sortBy(lambda pair: -1*pair[1]))
```

Example: Word count – Use of transform

```
# Print the result on the standard output
wordsCountsSortByKey pprint()

# Store the result in HDFS
wordsCountsSortByKey saveAsTextFiles(outputPathPrefix, "")

#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

Spark Structured Streaming

What is Spark Structured Streaming?

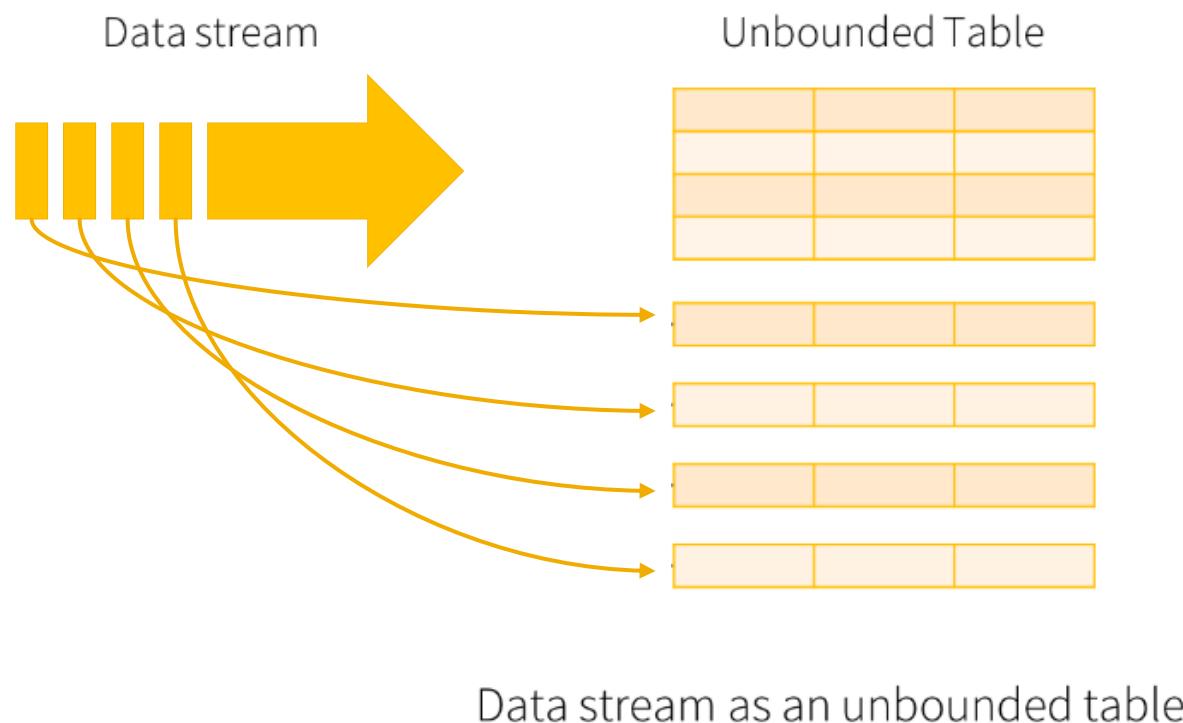
- **Structured Streaming** is a scalable and fault-tolerant stream processing engine that **is built on the Spark SQL engine**
- **Input data** are represented by means of (streaming) **DataFrames**
- Structured Streaming uses the existing Spark SQL APIs to query data streams
 - The same methods we used for analyzing “static” DataFrames
- A set of specific methods that are used to define
 - Input and output streams
 - Windows

Input data model

- Each input data stream is modeled as a table that is being continuously appended
 - Every time new data arrive they are appended at the end of the table
 - i.e., each **data stream** is considered an **unbounded input table**

Input data model

- New input data in the stream = new rows appended to an unbounded table



Queries

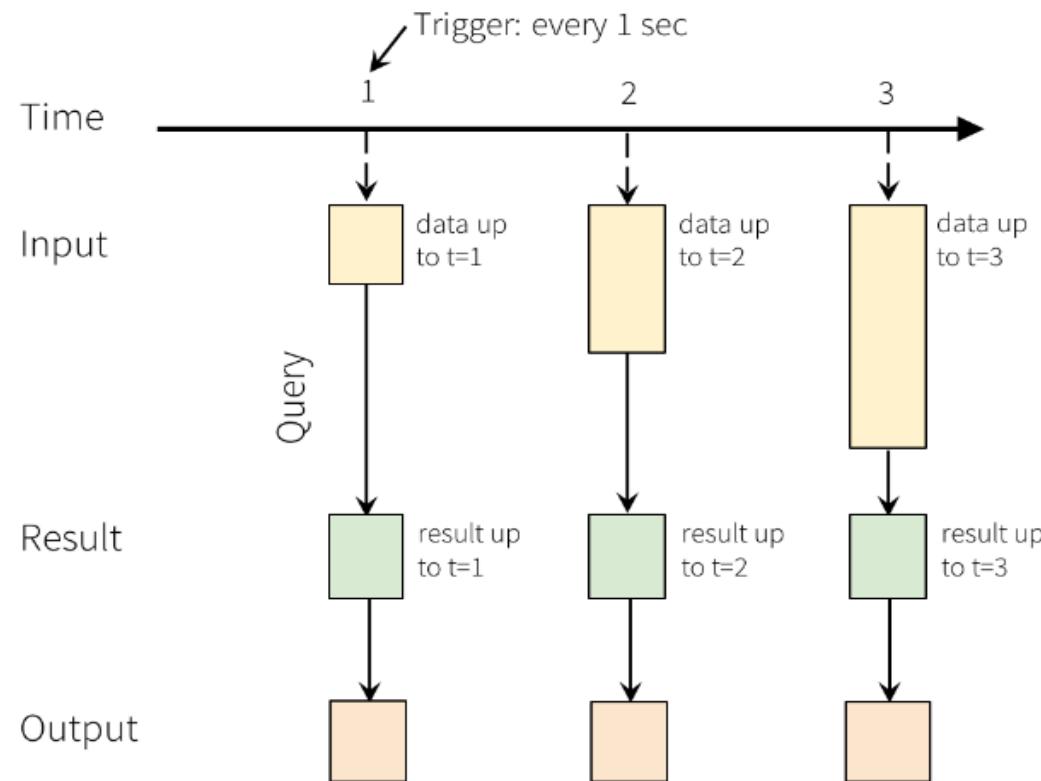
- The expressed queries are incremental queries that are run incrementally on the unbounded input tables
 - The arrival of new data triggers the execution of the incremental queries
 - The **result of a query** at a specific timestamp is the one obtained by running the query **on all the data arrived until that timestamp**
 - i.e., “stateful queries” are executed
 - Aggregation queries combine new data with the previous results to optimize the computation of the new results

Queries

- The queries can be executed
 - As micro-batch queries with a fixed batch interval
 - **Standard behavior**
 - **Exactly-once** fault-tolerance guarantees
 - As continuous queries
 - **Experimental**
 - **At-least-once** fault-tolerance guarantees

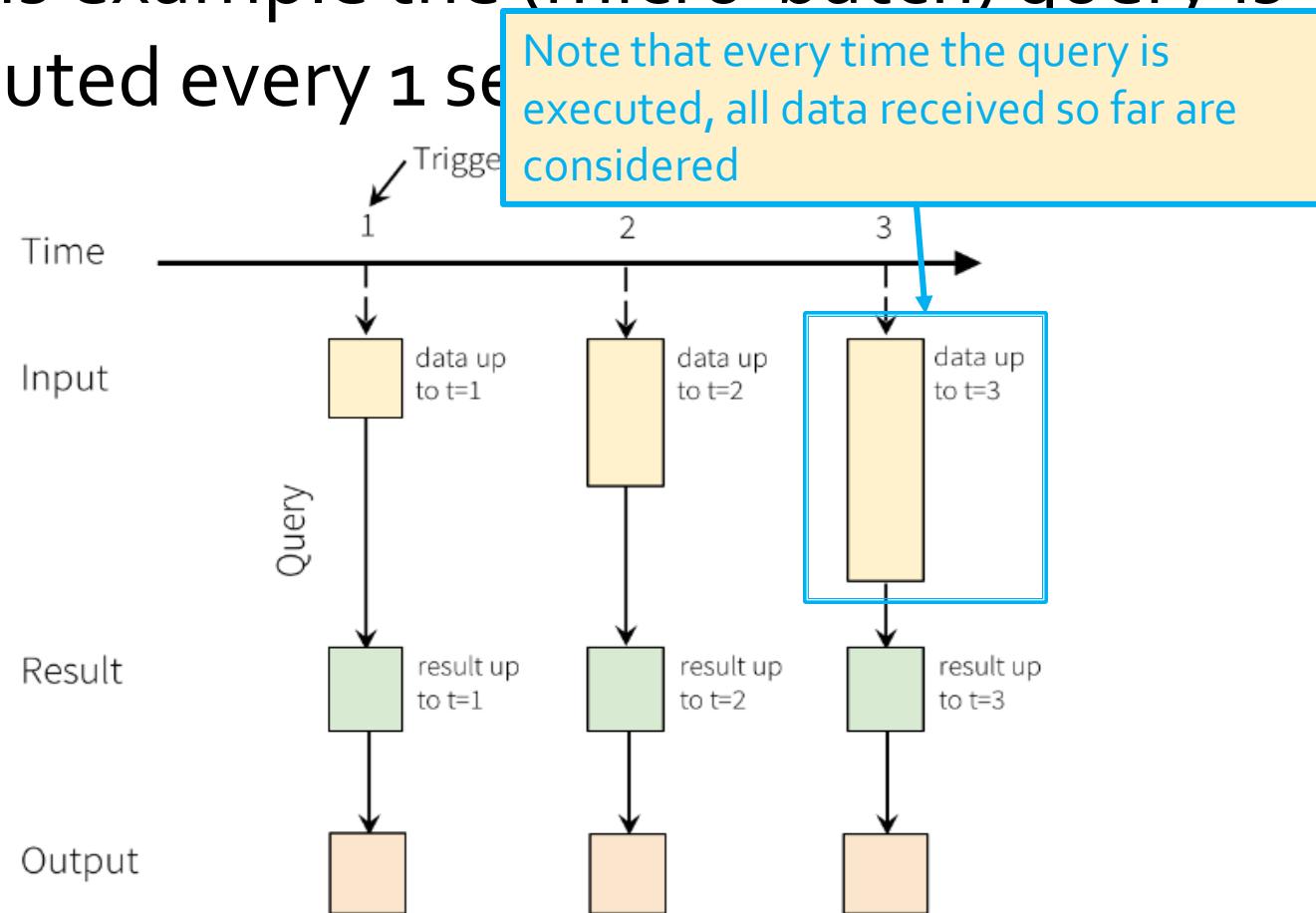
Queries

- In this example the (micro-batch) query is executed every 1 second



Queries

- In this example the (micro-batch) query is executed every 1 second

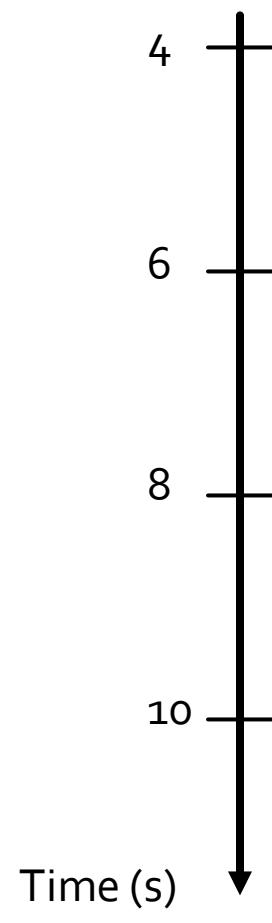


Example

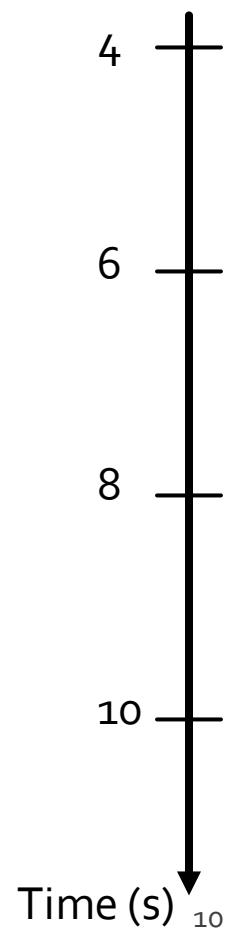
- Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
- For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - Print the requested information when new data are received by using the micro-batch processing mode
 - Suppose the batch-duration is set to 2 seconds

Example

Input stream

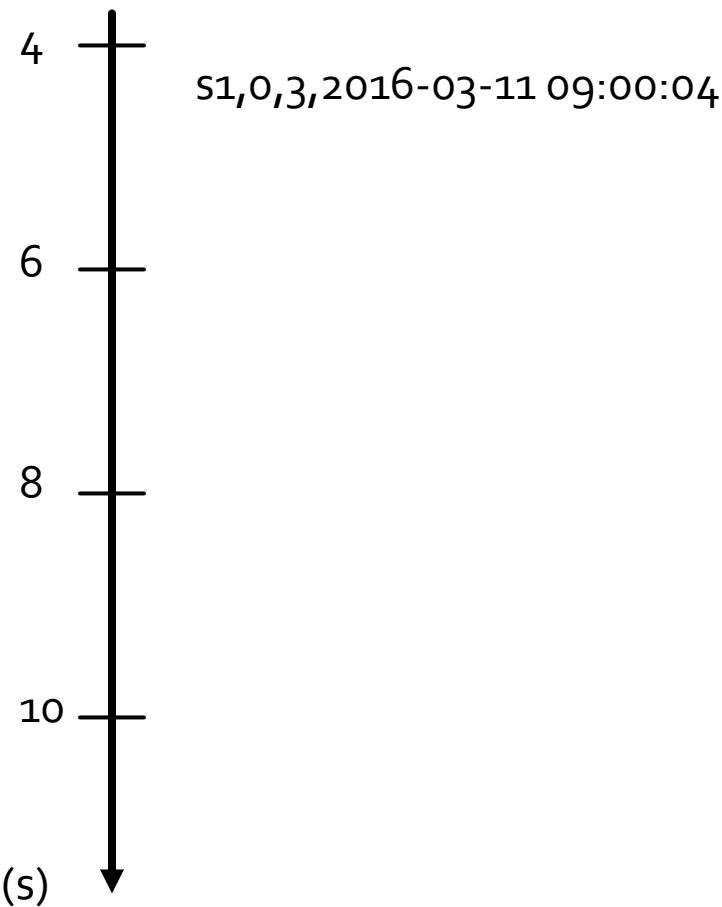


Console stdout

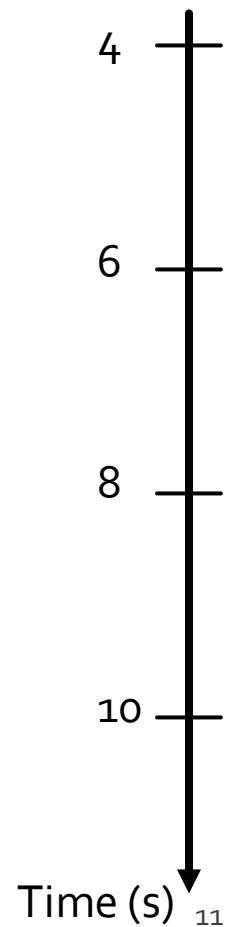


Example

Input stream

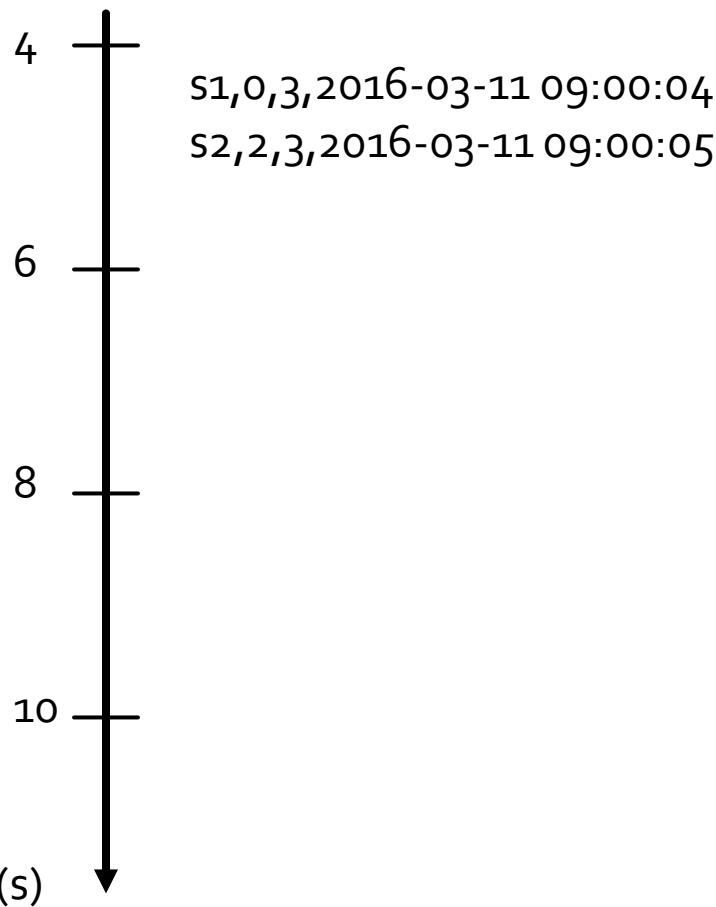


Console stdout

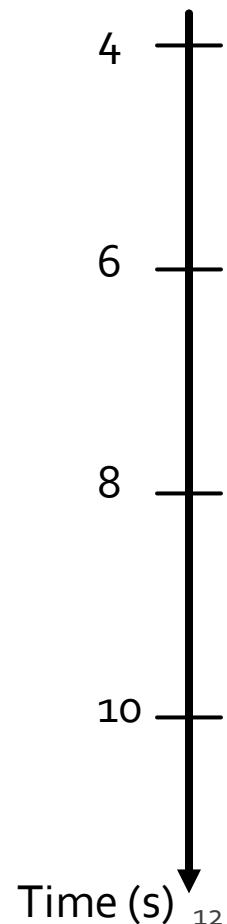


Example

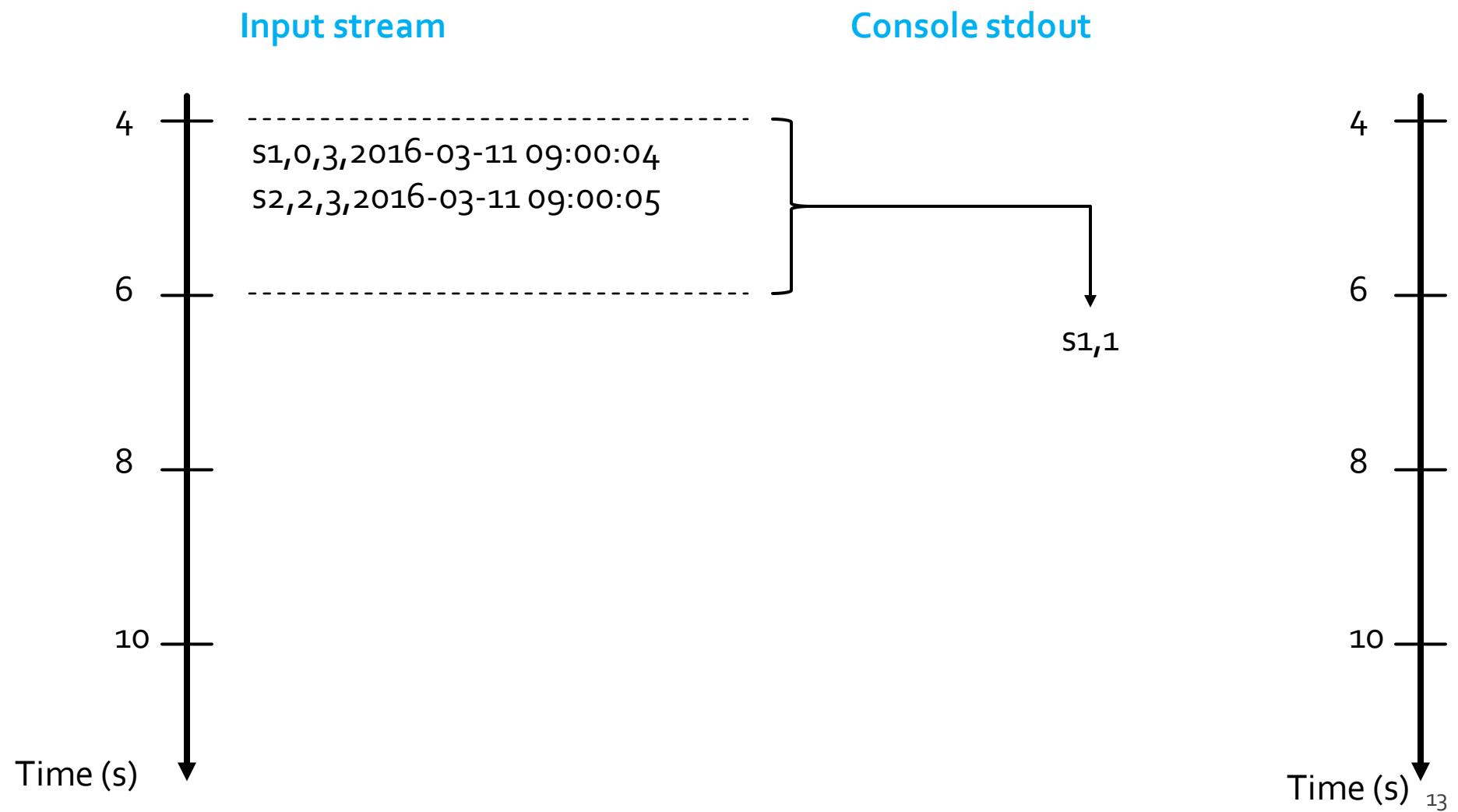
Input stream



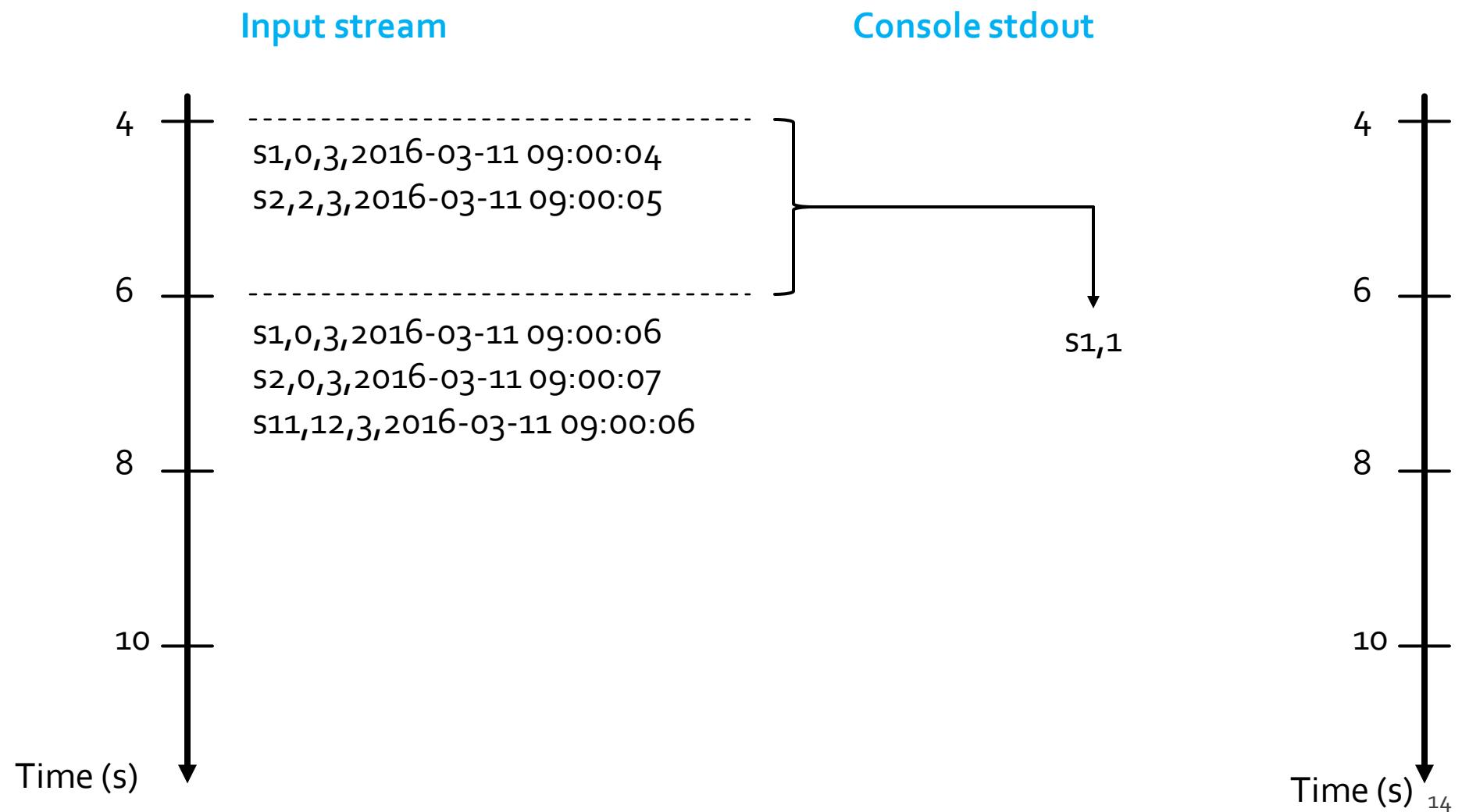
Console stdout



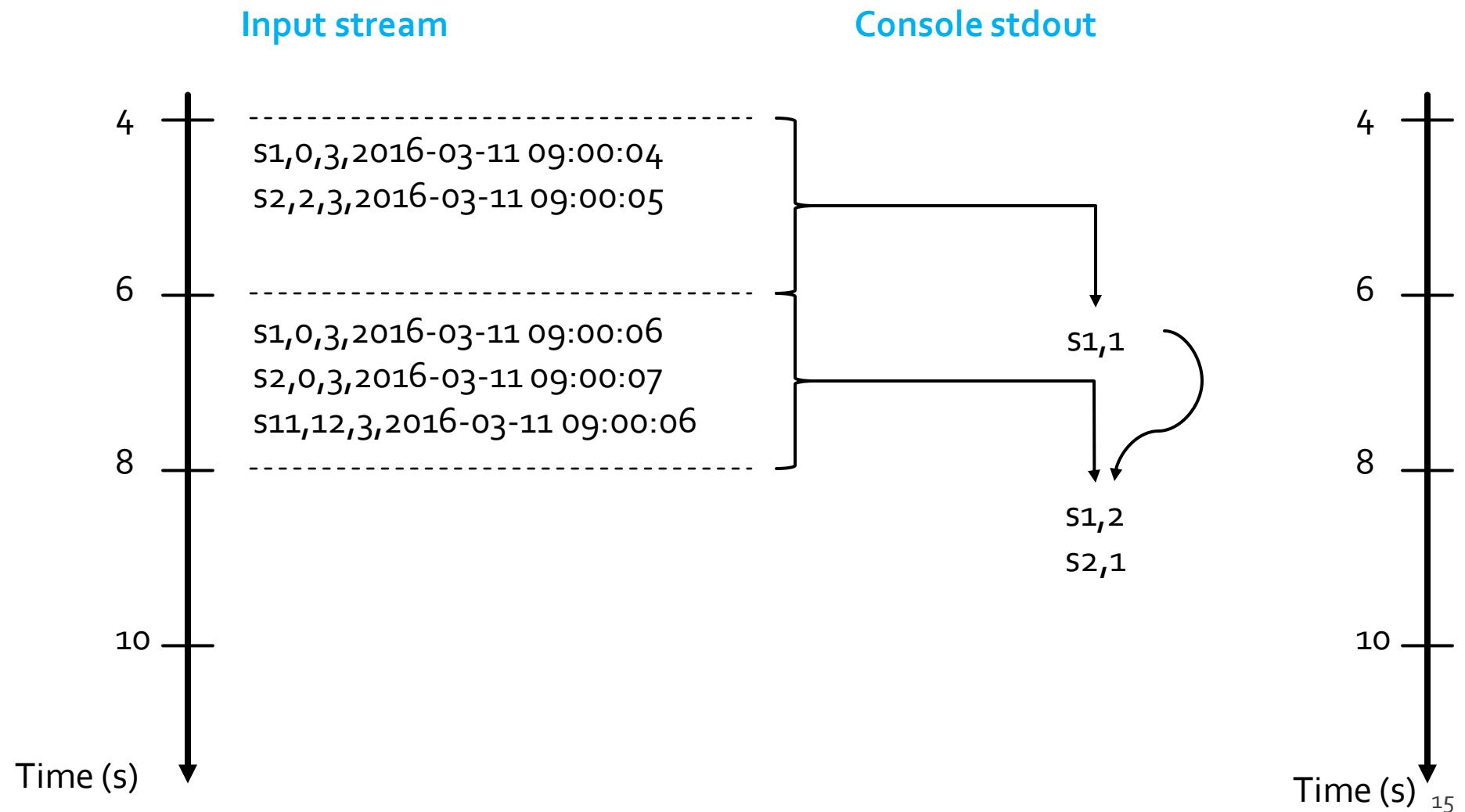
Example



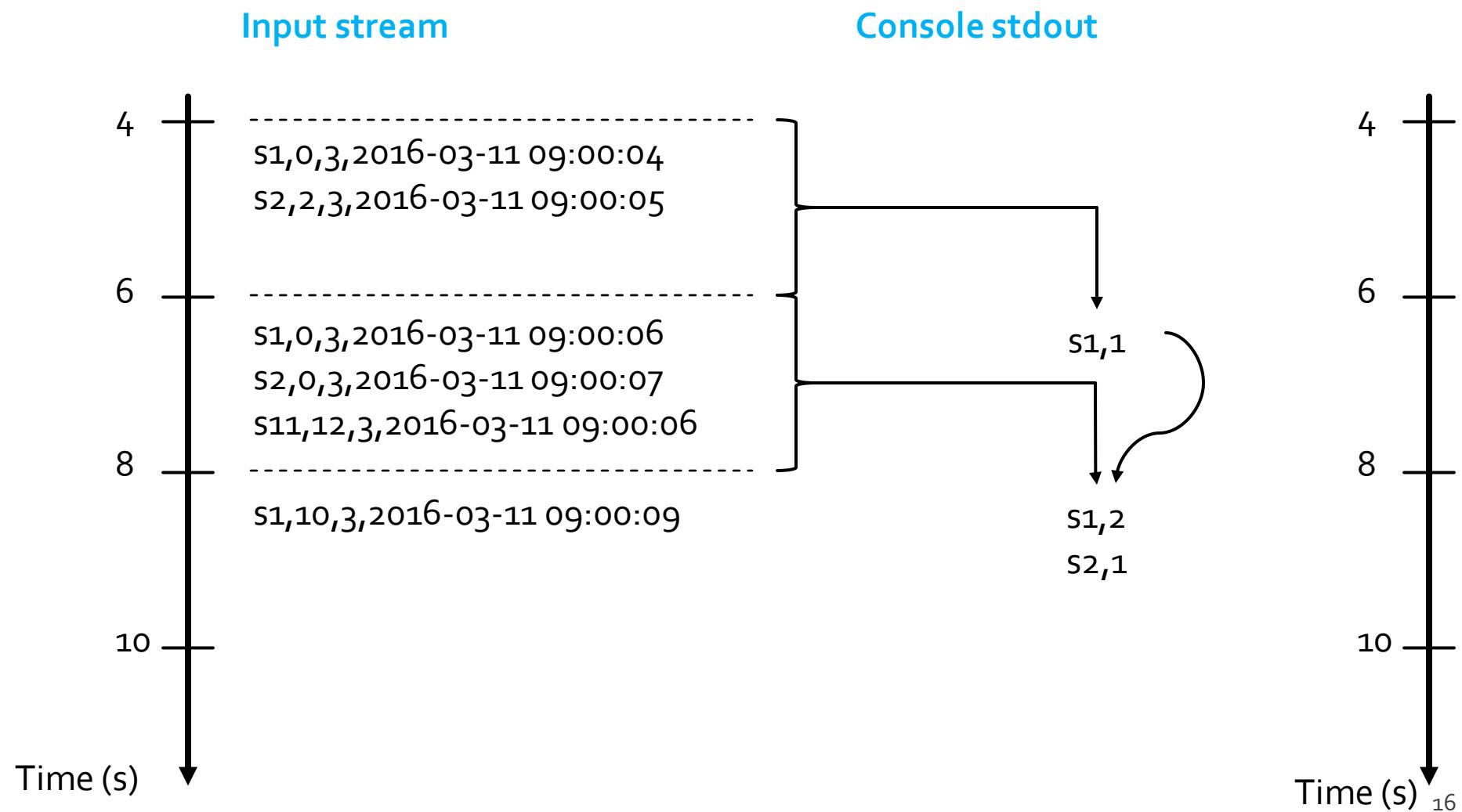
Example



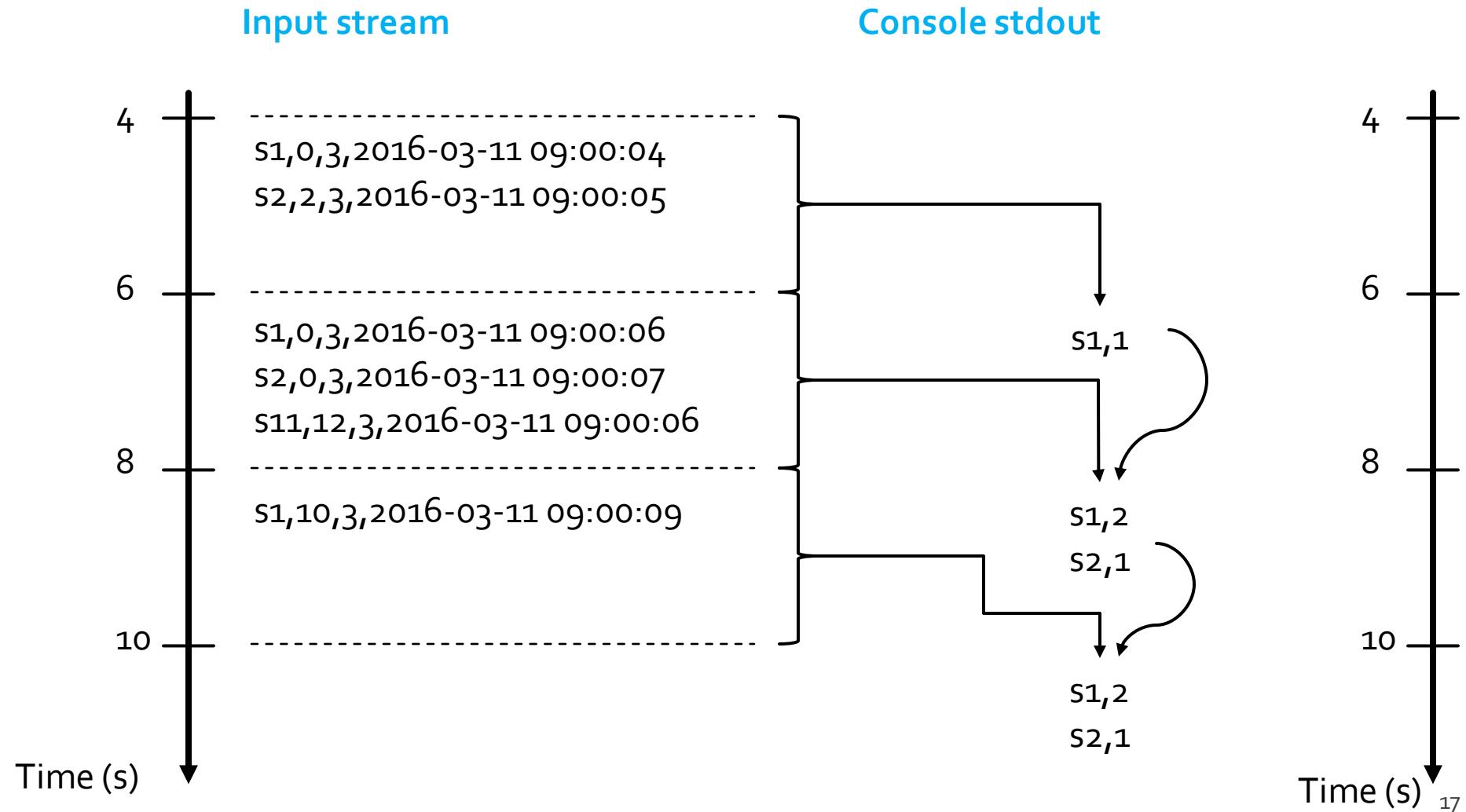
Example



Example



Example



Key concepts

- Input sources
- Transformations
- Outputs
 - External destinations/sinks
 - Output Modes
- Query run/execution
- Triggers

Input sources

- File source
 - Reads files written in a directory as a stream of data
 - Each line of the input file is an input record
 - Supported file formats are text, csv, json, orc, parquet, ..
- Kafka source
 - Reads data from Kafka
 - Each Kafka message is one input record

Input sources

- Socket source (for debugging purposes)
 - Reads UTF8 text data from a socket connection
 - This type of source **does not provide end-to-end fault-tolerance guarantees**
- Rate source (for debugging purposes)
 - Generates data at the specified number of rows per second
 - Each generated row contains a timestamp and value of type long

Input sources

- The `readStream` property of the `SparkSession` class is used to create `DataStreamReader`s
- The methods `format()` and `option()` of the `DataStreamReader` class are used to specify the input streams
 - Type, location, ...
- The method `load()` of the `DataStreamReader` class is used to return `DataFrames` associated with the input data streams

Input sources

- In this example the (streaming) DataFrame recordsDF is created and associated with the input stream of type socket
 - Address: localhost
 - Input port: 9999

```
recordsDF = spark.readStream \  
  .format("socket") \  
  .option("host", "localhost") \  
  .option("port", 9999) \  
  .load()
```

Transformations

- Transformations are the same of DataFrames
- However, there are **restrictions on** some types of queries/**transformations that cannot be executed incrementally**

Transformations

- **Unsupported** operations
 - **Multiple streaming aggregations** (i.e. a chain of aggregations on a streaming DataFrame)
 - **Limit and take first N rows**
 - **Distinct** operations
 - **Sorting** operations are supported on streaming DataFrames **only after** an aggregation and in complete output mode
 - **Few types of outer joins** on streaming DataFrames are not supported

Outputs

- Sinks
 - They are instances of the class DataStreamWriter and are used to specify the external destinations and store the results in the external destinations
- File sink
 - Stores the output to a directory
 - Supported file formats are text, csv, json, orc, parquet, ..
- Kafka sink
 - Stores the output to one or more topics in Kafka
- Foreach sink
 - Runs arbitrary computation on the output records

Outputs

- Console sink (for debugging purposes)
 - Prints the computed output to the console every time a new batch of records has been analyzed
 - This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after each computation
- Memory sink (for debugging purposes)
 - The output is stored in memory as an in-memory table
 - This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory

Output modes

- We must define how we want Spark to write output data in the external destinations
- Supported output modes:
 - Append
 - Complete
 - Update
- The supported output modes depend on the query type

Output modes

- Append mode
 - Default mode
 - Only the new rows added to the computed result since the last trigger (computation) will be outputted
 - This mode is supported for only those queries where rows added to the result is never going to change
 - This mode guarantees that each row will be output only once
 - Queries with only select, filter, map, flatMap, filter, join, etc. support append mode

Output modes

- Complete mode
 - The whole computed result will be outputted to the sink after every trigger (computation)
 - This mode is supported for aggregation queries

Output modes

- Update mode
 - Only the rows in the computed result that were updated since the last trigger (computation) will be outputted
- The complete list of supported output modes for each query type is available at
 - <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-modes>

Outputs

- The `writeStream` property of the `SparkSession` class is used to create `DataStreamWriter`s
- The methods `outputMode()`, `format()` and `option()` of the `DataStreamWriter` class are used to specify the output destination
 - Data format, location, output mode, etc.

Outputs

- In this example
 - The DataStreamWriter streamWriterRes is created and associated with the console
 - The output mode is set to append

```
streamWriterRes = stationIdTimestampDF \  
 .writeStream \  
 .outputMode("append") \  
 .format("console")
```

Query run/execution

- To start executing the defined queries/structured streaming applications you must explicitly invoke the `start()` action on the defined sinks (DataStreamWriter objects associated with the external destinations in which the results will be stored)
- You can start several queries in the same application
- Structured streaming queries run forever
 - You must explicitly stop/kill them otherwise they will run forever

Triggers

- For each Spark structured streaming query we can specify when new input data must be processed
- And whether the query is going to be executed
 - as a micro-batch query with a fixed batch interval
 - or as a continuous processing query (experimental)
- The trigger type for each query is specified by means of the **trigger()** method of the DataStreamWriter class

Trigger Types

- No trigger type is explicitly specified
 - Default trigger setting
 - The query will be executed in **micro-batch mode**
 - Each micro-batch is generated and processed as soon as the previous micro-batch has been processed

Trigger Types

- Fixed interval micro-batches
 - The query will be executed in **micro-batch mode**
 - Micro-batches will be processed at the user-specified intervals
 - The parameter `processingTime` of the trigger method() is used to specify the micro-batch size
 - If the previous micro-batch completes within its interval, then the engine will wait until the interval is over before processing the next micro-batch

Trigger Types

- If the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes

Trigger Types

- One-time micro-batch
 - The query will be executed in **micro-batch mode**
 - But the query will be **executed only one time** on one single micro-batch containing all the available data of the input stream
 - After the single execution the query stops on its own

Trigger Types

- This trigger type is useful when you want to periodically spin up a cluster, process everything that is available since the last period, and then shutdown the cluster
 - In some case, this may lead to significant cost savings

Trigger Types

- Continuous with fixed checkpoint interval
(experimental)
 - The query will be executed in the new low-latency,
continuous processing mode
 - **At-least-once** fault-tolerance guarantees

Sparks Structured Streaming: Example 1

- Problem specification
 - Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
 - Output
 - For each input reading with a number of free slots equal to 0 print on the standard output the value of stationId and timestamp
 - Use the standard micro-batch processing mode

Sparks Structured Streaming: Example 1

```
from pyspark.sql.types import *
from pyspark.sql.functions import split

# Create a "receiver" DataFrame that will connect to localhost:9999
recordsDF = spark.readStream\
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

Sparks Structured Streaming: Example 1

```
# The input records are characterized by one single column called value
# of type string
# Example of an input record: s1,0,3,2016-03-11 09:00:04
# Define four more columns by splitting the input column value
# New columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF\
.withColumn("stationId", split(recordsDF.value,',')[0].cast("string"))\
.withColumn("freeslots", split(recordsDF.value,',')[1].cast("integer"))\
.withColumn("usedslots", split(recordsDF.value,',')[2].cast("integer"))\
.withColumn("timestamp", split(recordsDF.value,',')[3].cast("timestamp"))
```

Sparks Structured Streaming: Example 1

```
# The input records are characterized by one single column called value  
# of type string
```

```
# Example of an input record: s1,0,3,2016-03-11 09:00:04
```

```
# Define four more columns by splitting the input column value
```

withColumn () is used to add new columns.

It is a standard DataFrame method.

withColumn () returns a DataFrame with the same columns
of the input DataFrame and the new defined column

```
readingsDF = recordsDF\  
.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))\  
.withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))\  
.withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))\  
.withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
```

Sparks Structured Streaming: Example 1

```
# The input records are characterized by one single column called value  
# of type string  
# Example of an input record: s1,0,3,2016-03-11 09:00:04  
# Define four more columns by splitting the input column value
```

For each new column you must specify:

- Name
- The SQL function that is used to define its value in each record

The cast() method is used to specify the data type of each defined column.

```
readingsDF = recordsDF\  
.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))\  
.withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))\  
.withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))\  
.withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
```

Sparks Structured Streaming: Example 1

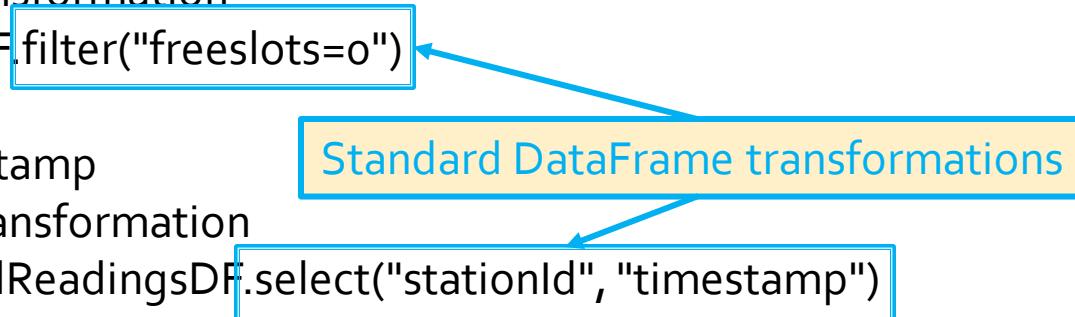
```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Select stationid and timestamp
# Use the standard select transformation
stationIdTimestampDF = fullReadingsDF.select("stationId", "timestamp")
```

Sparks Structured Streaming: Example 1

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")  
  
# Select stationid and timestamp
# Use the standard select transformation
stationIdTimestampDF = fullReadingsDF.select("stationId", "timestamp")
```

Standard DataFrame transformations



Sparks Structured Streaming: Example 1

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Select stationid and timestamp
# Use the standard select transformation
stationIdTimestampDF = fullReadingsDF.select("stationId", "timestamp")

# The result of the structured streaming query will be stored/printed on
# the console "sink".
# append output mode
queryFilterStreamWriter = stationIdTimestampDF \
.writeStream \
.outputMode("append") \
.format("console")
```

Sparks Structured Streaming: Example 1

```
# Start the execution of the query (it will be executed until it is explicitly stopped)
queryFilter = queryFilterStreamWriter.start()
```

Sparks Structured Streaming: Example 2

- Problem specification
 - Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
 - Output
 - For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - Print the requested information when new data are received by using the standard micro-batch processing mode

Sparks Structured Streaming: Example 2

```
from pyspark.sql.types import *
from pyspark.sql.functions import split

# Create a "receiver" DataFrame that will connect to localhost:9999
recordsDF = spark.readStream\
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

Sparks Structured Streaming: Example 2

```
# The input records are characterized by one single column called value
# of type string
# Example of an input record: s1,0,3,2016-03-11 09:00:04
# Define four more columns by splitting the input column value
# New columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF\
.withColumn("stationId", split(recordsDF.value,',')[0].cast("string"))\
.withColumn("freeslots", split(recordsDF.value,',')[1].cast("integer"))\
.withColumn("usedslots", split(recordsDF.value,',')[2].cast("integer"))\
.withColumn("timestamp", split(recordsDF.value,',')[3].cast("timestamp"))
```

Sparks Structured Streaming: Example 2

```
# Filter data  
# Use the standard filter transformation  
fullReadingsDF = readingsDF.filter("freeslots=0")
```

Sparks Structured Streaming: Example 2

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for each stationId
# The standard groupBy method is used
countsDF = fullReadingsDF\
    .groupBy("stationId")\
    .agg({"*": "count"})
```

Sparks Structured Streaming: Example 2

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for each stationId
# The standard groupBy method is used
countsDF = fullReadingsDF\
    .groupBy("stationId")\
    .agg({"*":"count"})
```

Standard DataFrame transformations

Sparks Structured Streaming: Example 2

```
# The result of the structured streaming query will be stored/printed on
# the console "sink"
# complete output mode
# (append mode cannot be used for aggregation queries)
queryCountStreamWriter = countsDF \
    .writeStream \
    .outputMode("complete") \
    .format("console")

# Start the execution of the query (it will be executed until it is explicitly stopped)
queryCount = queryCountStreamWriter.start()
```

Event Time and Window Operations

Event Time and Window Operations

- Input streaming records are usually characterized by a time information
 - It is the **time when the data was generated**
 - It is usually called **event-time**
- For many applications, you want to operate by taking into consideration the event-time and windows containing data associated with the same event-time range

Event Time and Window Operations

- For example
 - Compute the number of events generated by each monitored IoT device every minute based on the event-time
 - For each window associated with one distinct minute consider only the data with an event-time inside that minute/window and compute the number of events for each IoT device
 - One computation for each minute/window
 - You want to use the time when the data was generated (i.e., the event-time) rather than the time Spark receives them

Event Time and Window Operations

- Spark allows defining **windows based on the time-event** input column
- And then apply aggregation functions over each window

Event Time and Window Operations

- For each structured streaming query on which you want to apply a window computation you must
 - Specify the name of the time-event column in the input (streaming) DataFrame
 - The characteristics of the (sliding) windows
 - windowDuration
 - slideDuration
 - Do not set it if you want non-overlapped windows, i.e., if you want to a slideDuration equal to windowDuration
- You can set different window characteristics for each query of your application

Event Time and Window Operations

- The `window(timeColumn, windowDuration, slideDuration=None)` function is used inside the standard `groupBy()` one to specify the characteristics of the windows
- Windows can be used only with queries that are applying aggregation functions

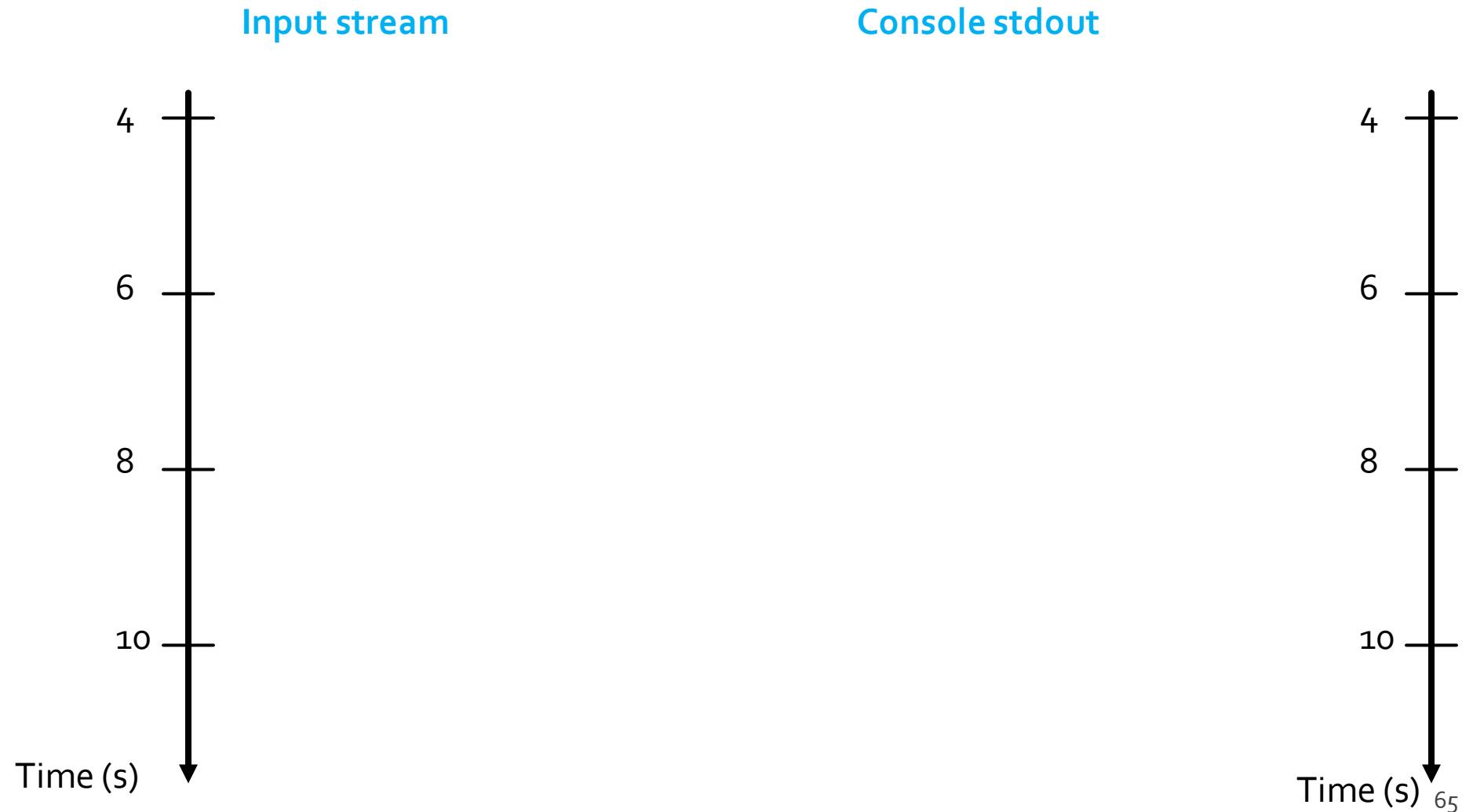
Event Time and Window Operations: Example 3

- Problem specification
 - Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
 - **timestamp** is the **event-time column**

Event Time and Window Operations: Example 3

- Output
 - For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0 in each window
 - The query is executed for each window
 - Set windowDuration to 2 seconds and no slideDuration
 - i.e., non-overlapped windows

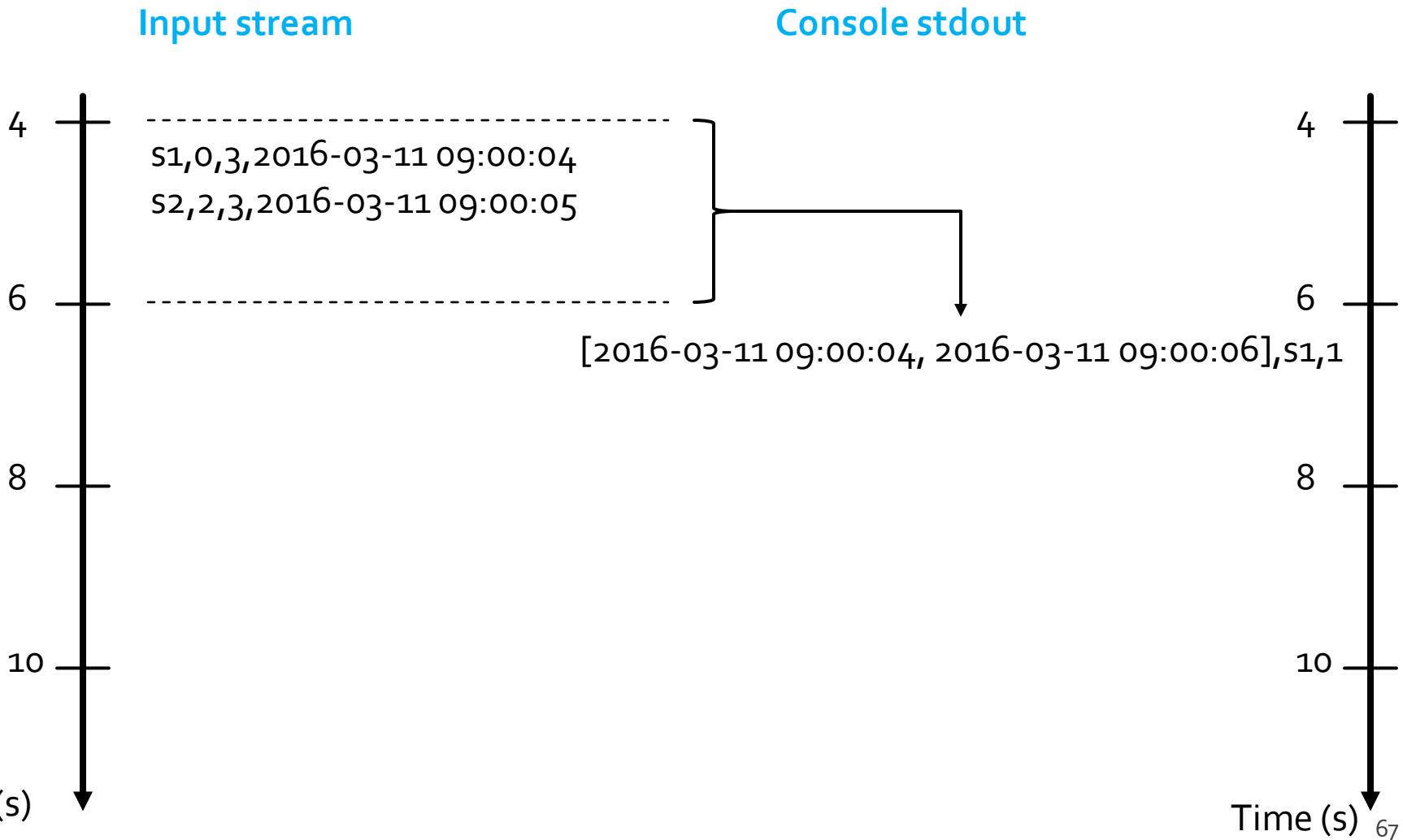
Event Time and Window Operations: Example 3



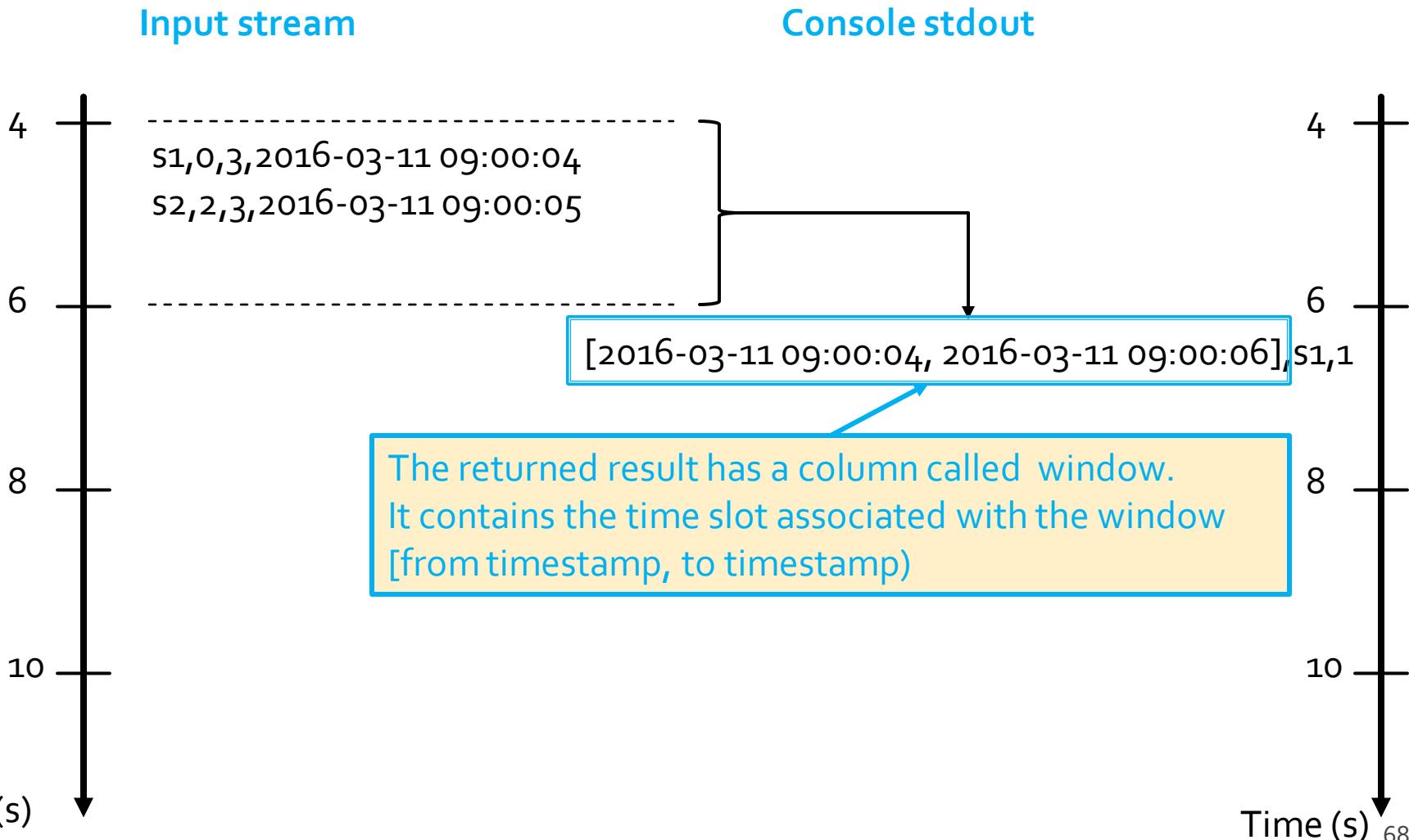
Event Time and Window Operations: Example 3



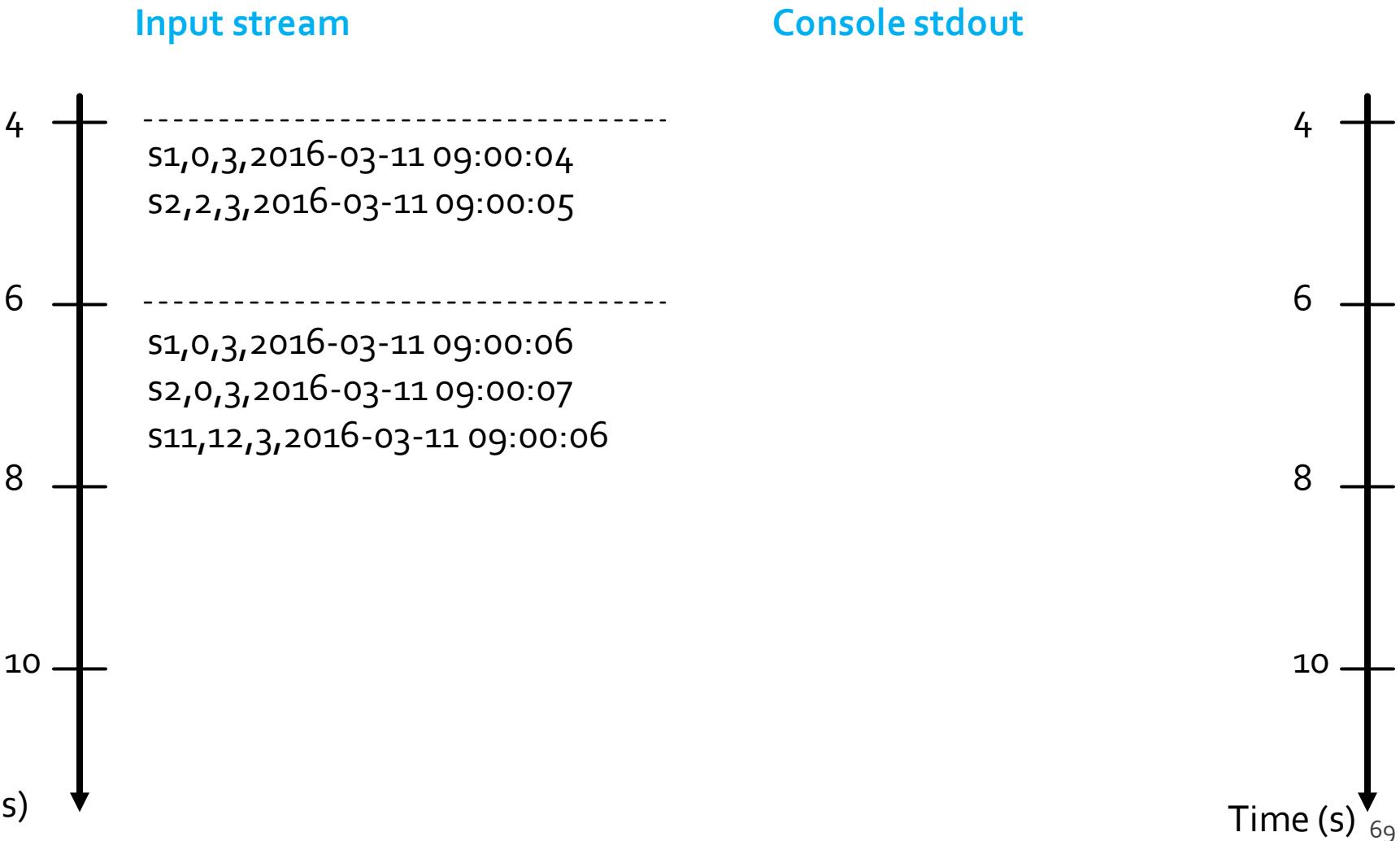
Event Time and Window Operations: Example 3



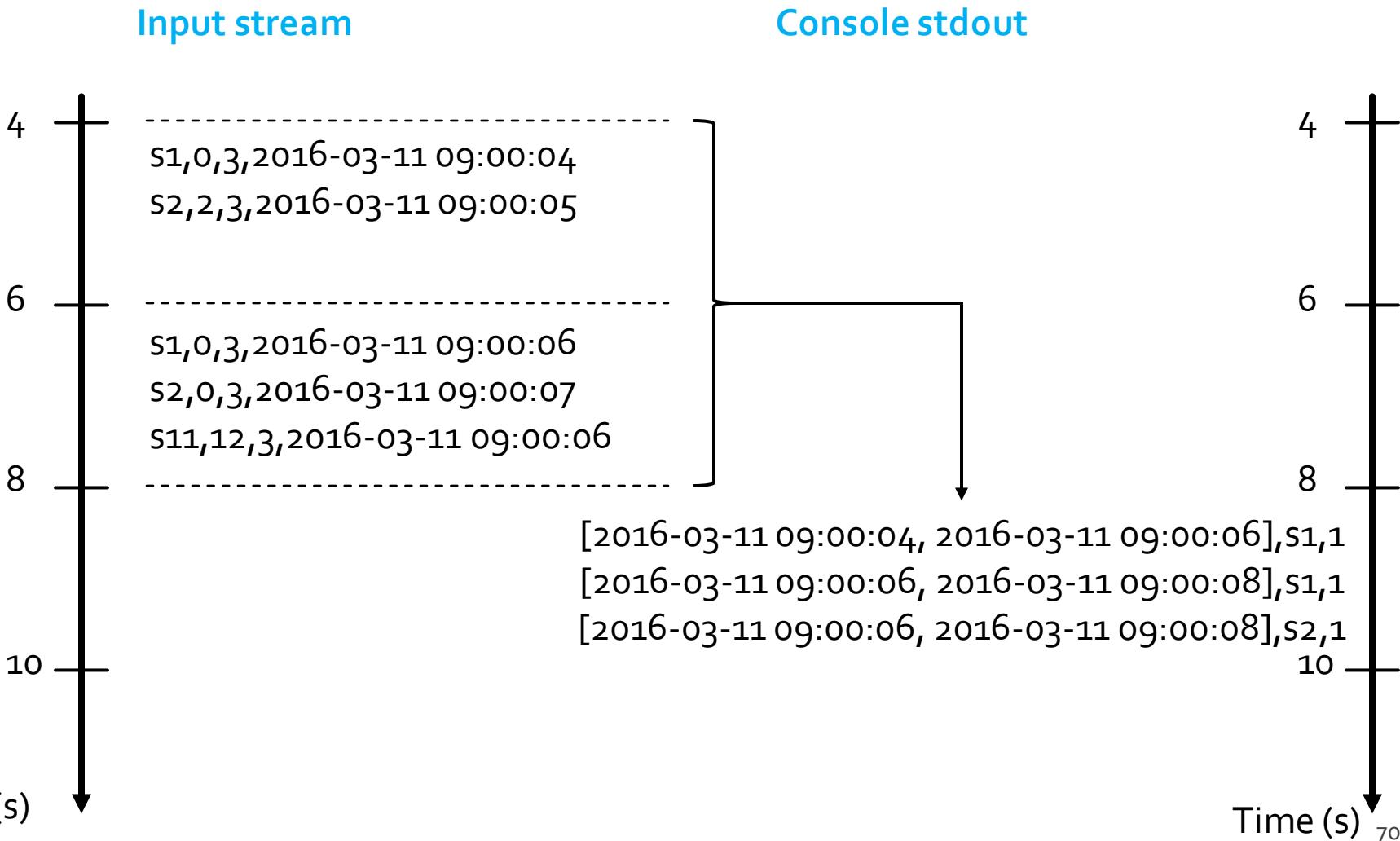
Event Time and Window Operations: Example 3



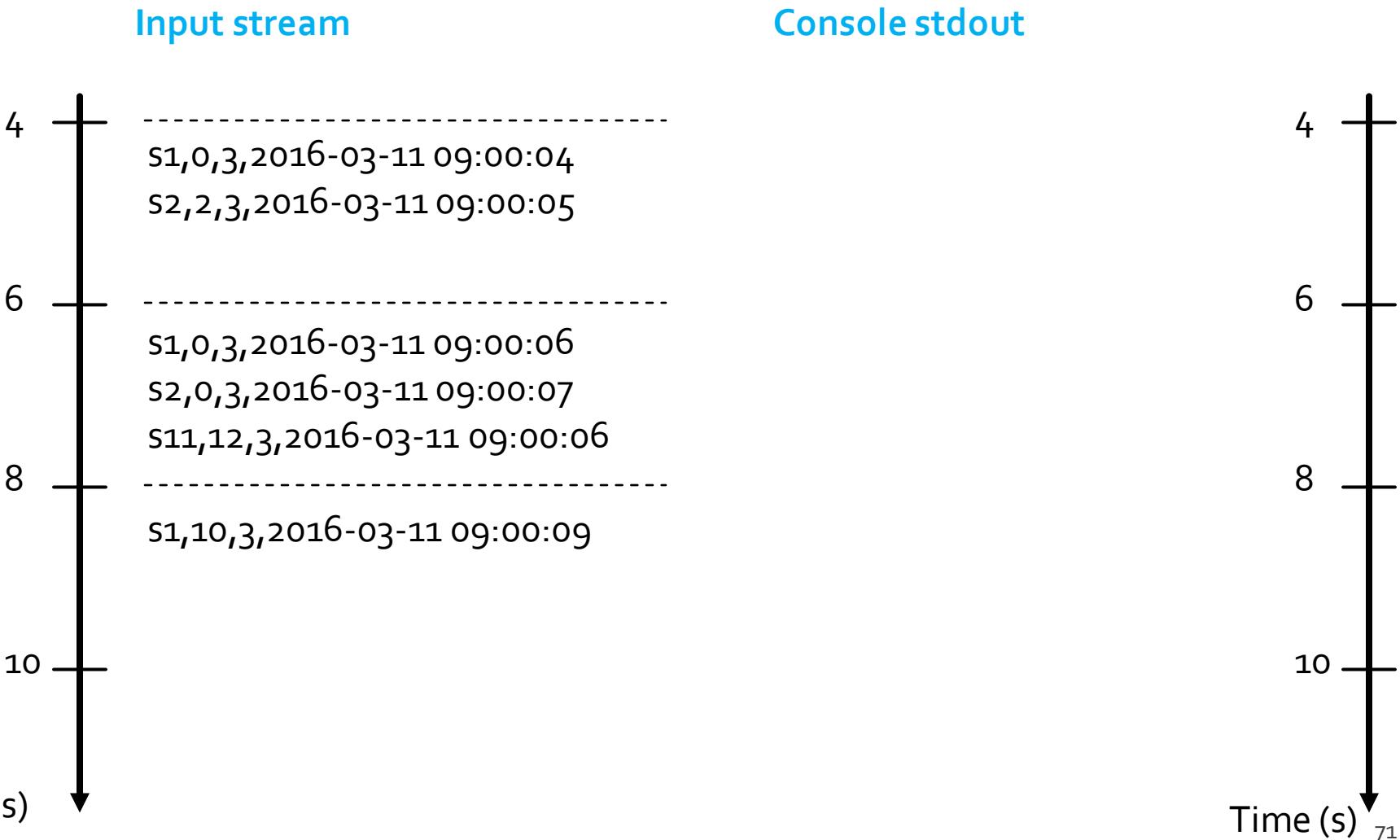
Event Time and Window Operations: Example 3



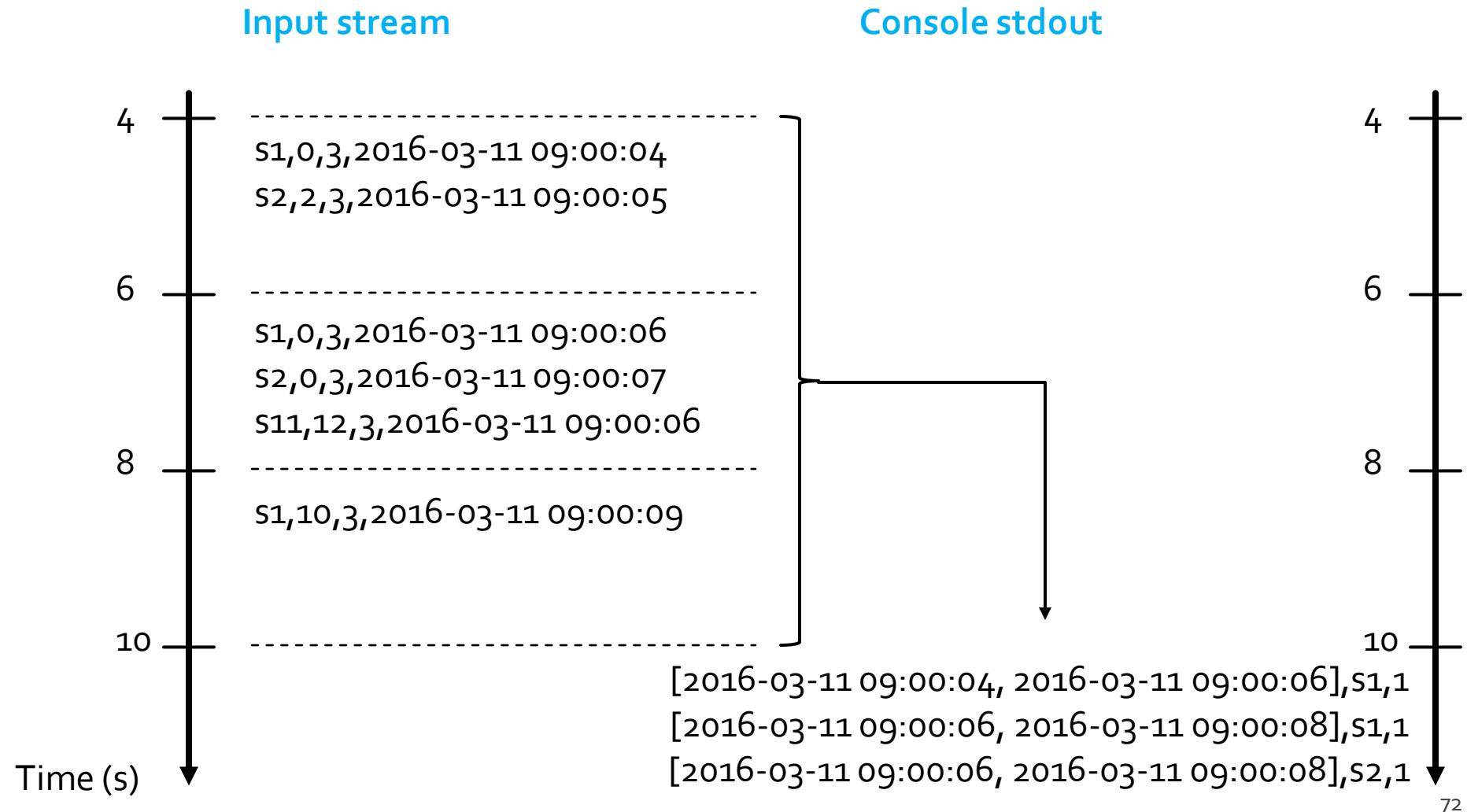
Event Time and Window Operations: Example 3



Event Time and Window Operations: Example 3



Event Time and Window Operations: Example 3



Event Time and Window Operations: Example 3

```
from pyspark.sql.types import *
from pyspark.sql.functions import split
from pyspark.sql.functions import window

# Create a "receiver" DataFrame that will connect to localhost:9999
recordsDF = spark.readStream\
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

Event Time and Window Operations: Example 3

```
# The input records are characterized by one single column called value
# of type string
# Example of an input record: s1,0,3,2016-03-11 09:00:04
# Define four more columns by splitting the input column value
# New columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF\
.withColumn("stationId", split(recordsDF.value,',')[0].cast("string"))\
.withColumn("freeslots", split(recordsDF.value,',')[1].cast("integer"))\
.withColumn("usedslots", split(recordsDF.value,',')[2].cast("integer"))\
.withColumn("timestamp", split(recordsDF.value,',')[3].cast("timestamp"))
```

Event Time and Window Operations: Example 3

```
# Filter data  
# Use the standard filter transformation  
fullReadingsDF = readingsDF.filter("freeslots=0")
```

Event Time and Window Operations: Example 3

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for each stationId in each window.
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF\
.groupBy(window(fullReadingsDF.timestamp, "2 seconds"), "stationId")\
.agg({"*":"count"})\
.sort("window")
```

Event Time and Window Operations: Example 3

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for each stationId in each window.
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF\
    .groupBy(window(fullReadingsDF.timestamp, "2 seconds"), "stationId")\
    .agg({"*": "count"})\
    .sort("window")
```

Specify window characteristics

Event Time and Window Operations: Example 3

```
# The result of the structured streaming query will be stored/printed on
# the console "sink"
# complete output mode
# (append mode cannot be used for aggregation queries)
queryCountWindowStreamWriter = countsDF \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option("truncate", "false")

# Start the execution of the query (it will be executed until it is explicitly stopped)
queryCountWindow = queryCountWindowStreamWriter.start()
```

Late data

- Sparks handles data that have arrived later than expected based on its event-time
 - They are called late data
- Spark has full control over updating old aggregates when there are late data
 - Every time new data are processed the result is computed by combining old aggregate values and the new data by considering the event-time column instead of the time Spark receives the data

Late data: Running example

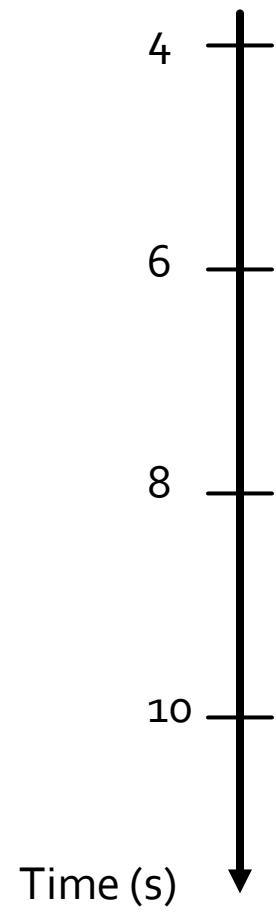
- Problem specification
 - Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
 - **timestamp** is the **event-time column**

Late data: Running example

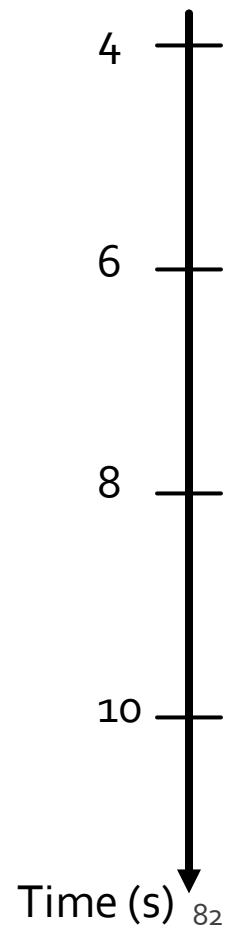
- Output
 - For each stationId, print on the standard output the total number of received input readings with a number of free slots equal to 0 in each window
 - The query is executed for each window
 - Set windowDuration to 2 seconds and no slideDuration
 - i.e., non-overlapped windows

Late data: Running example

Input stream

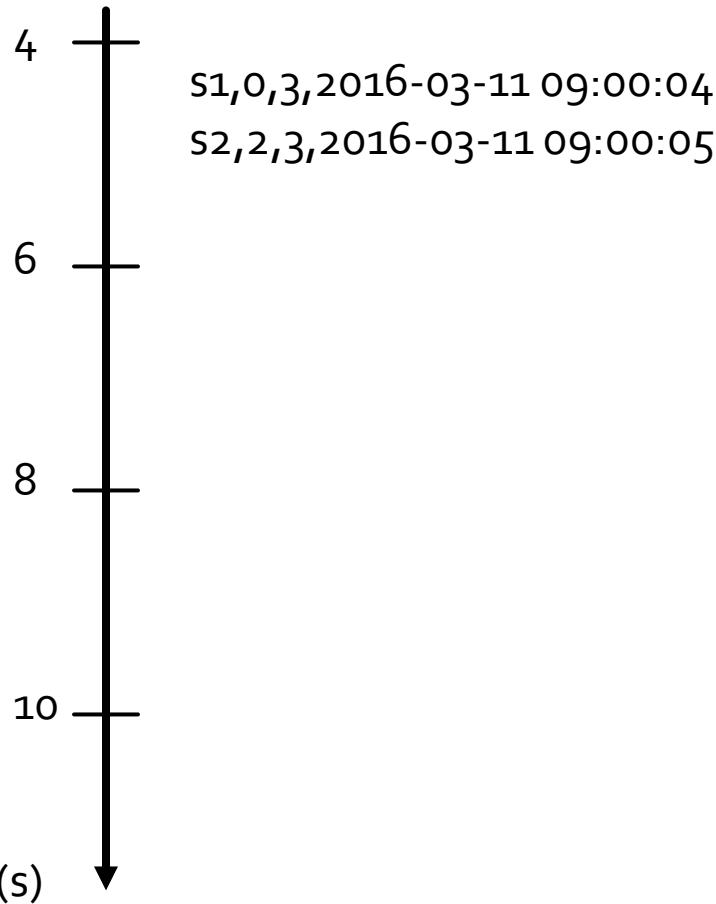


Console stdout

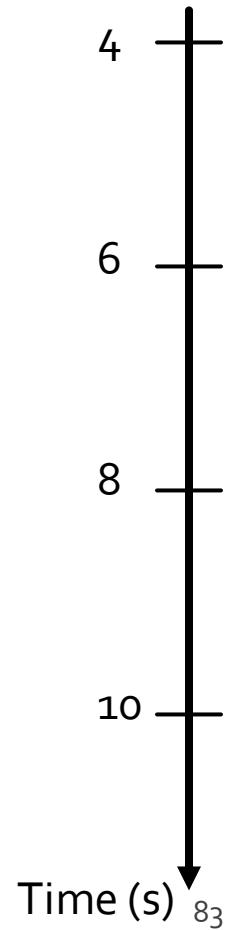


Late data: Running example

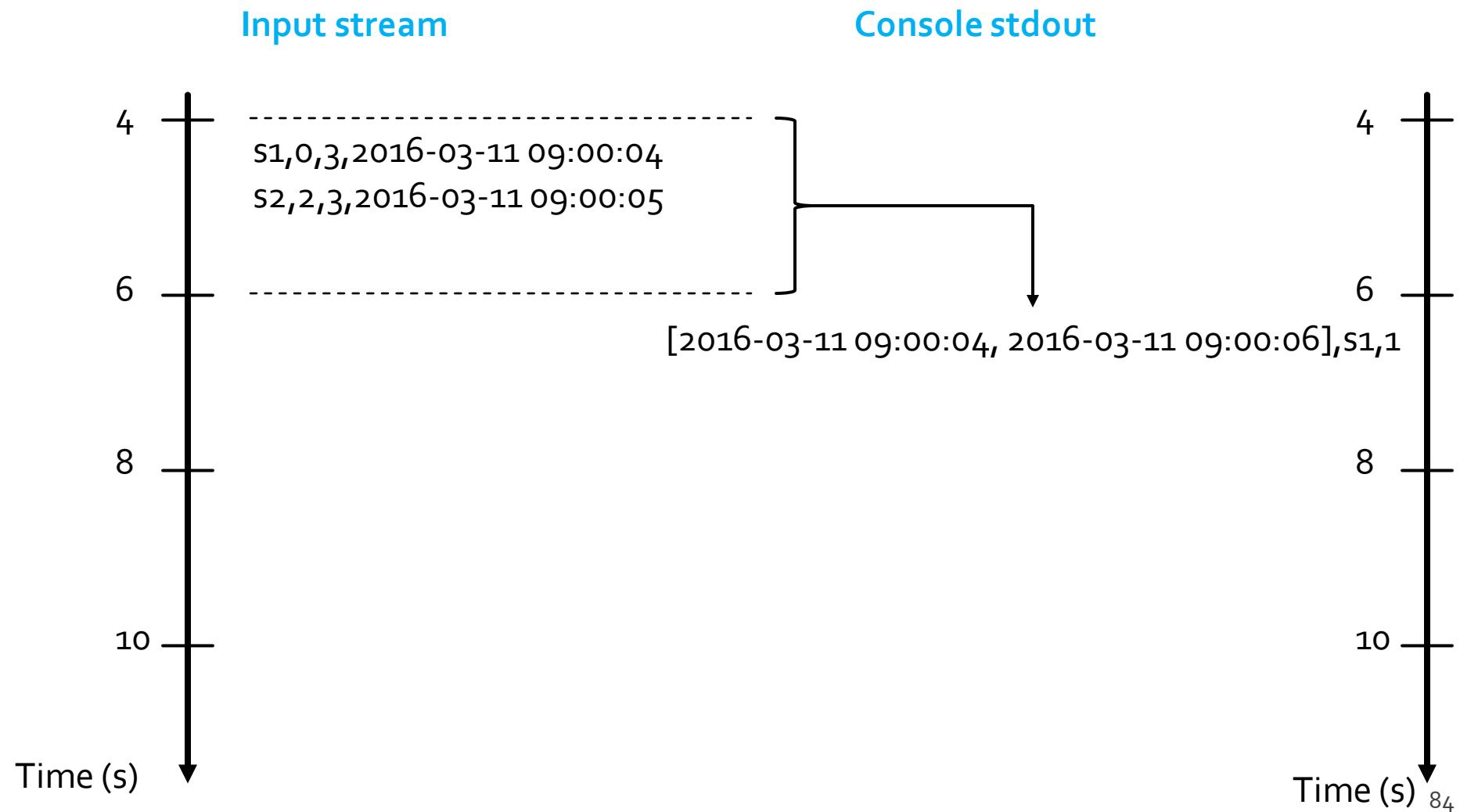
Input stream



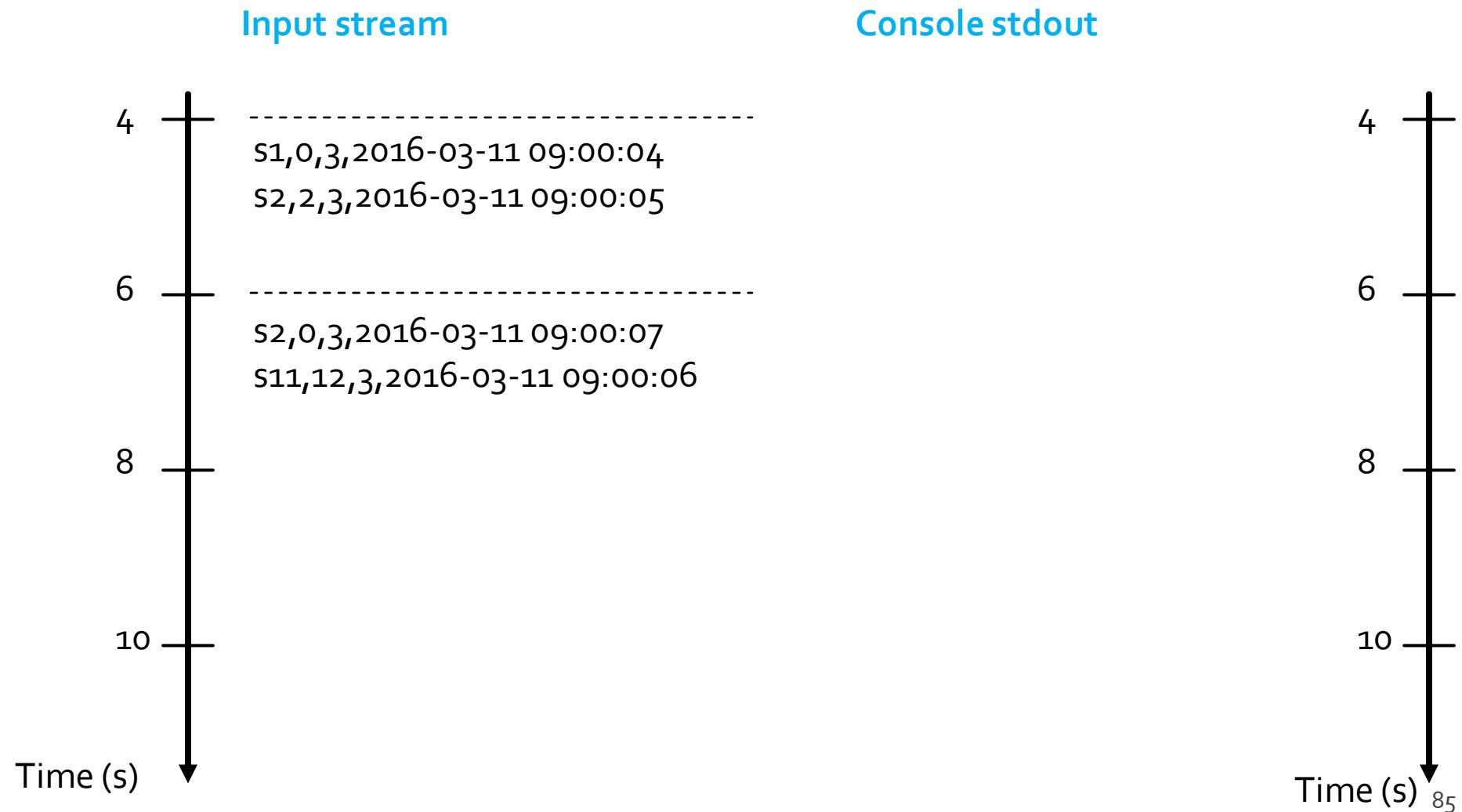
Console stdout



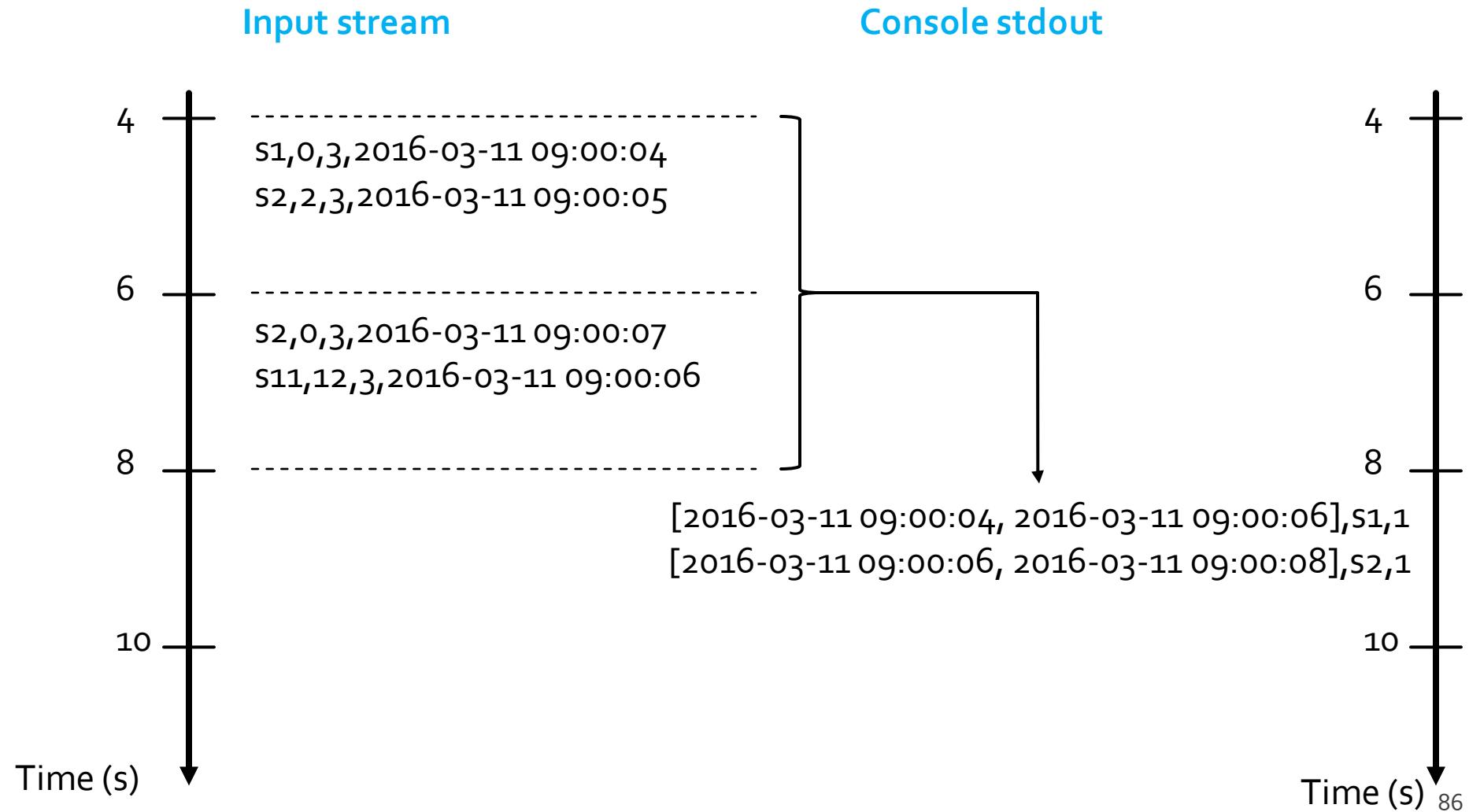
Late data: Running example



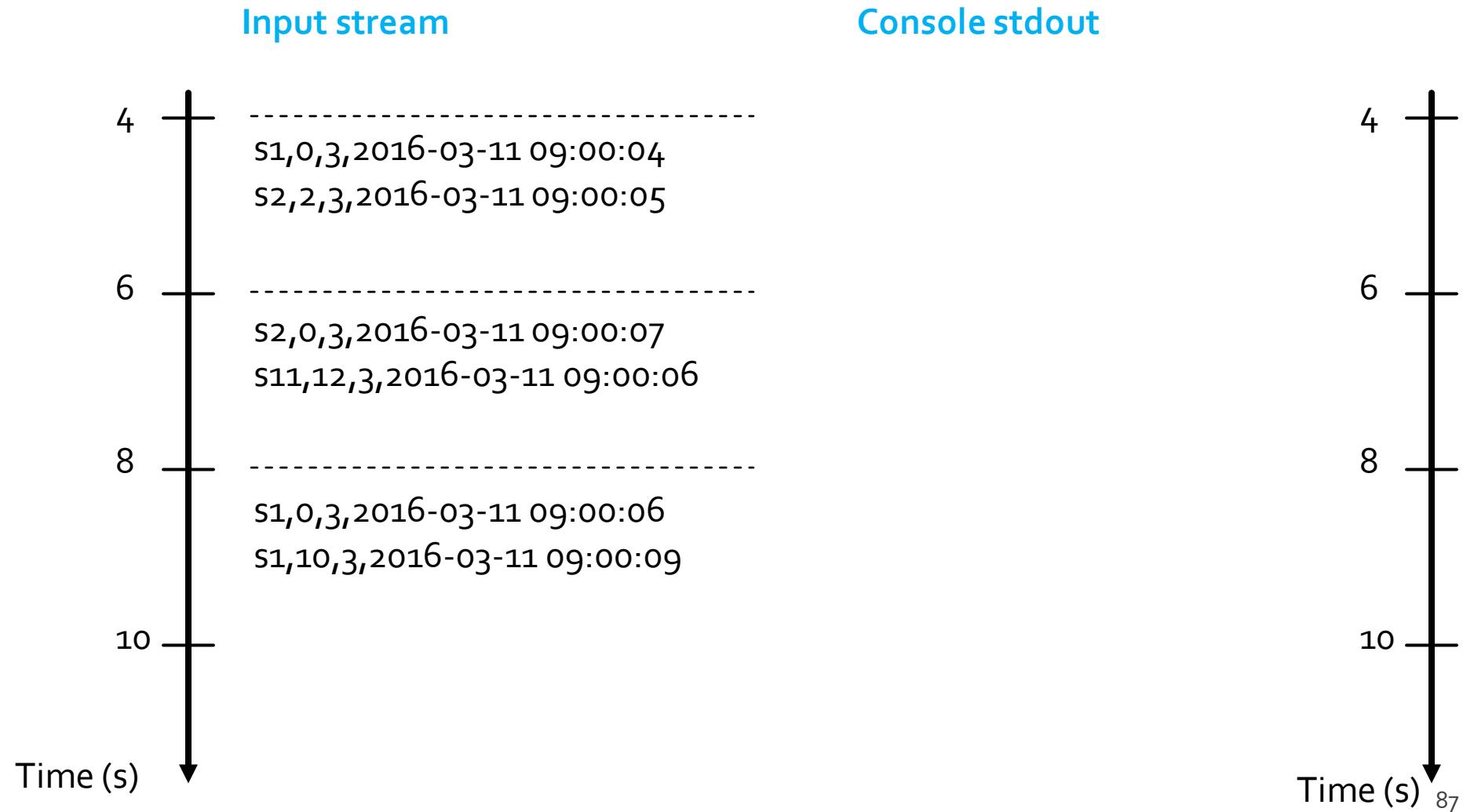
Late data: Running example



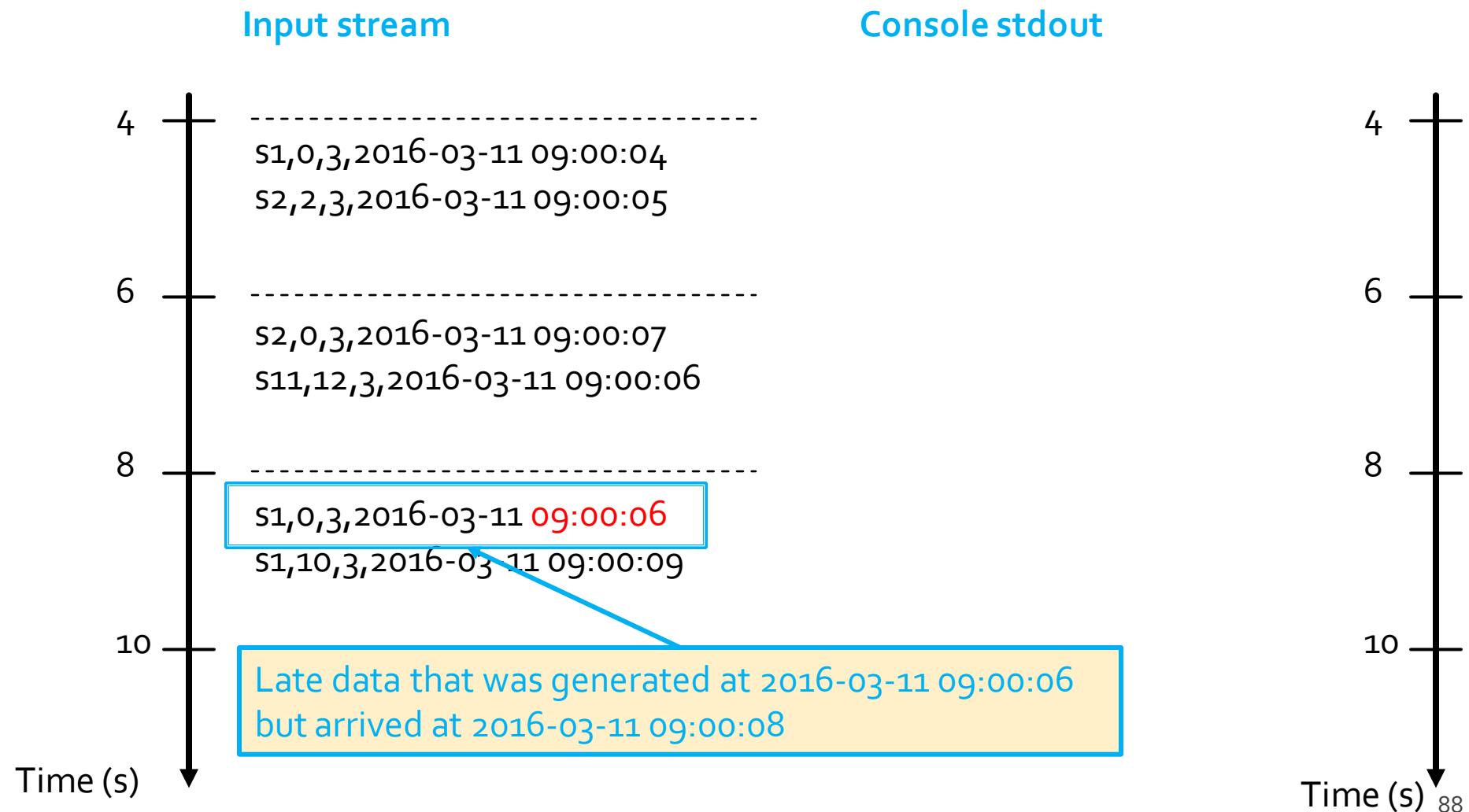
Late data: Running example



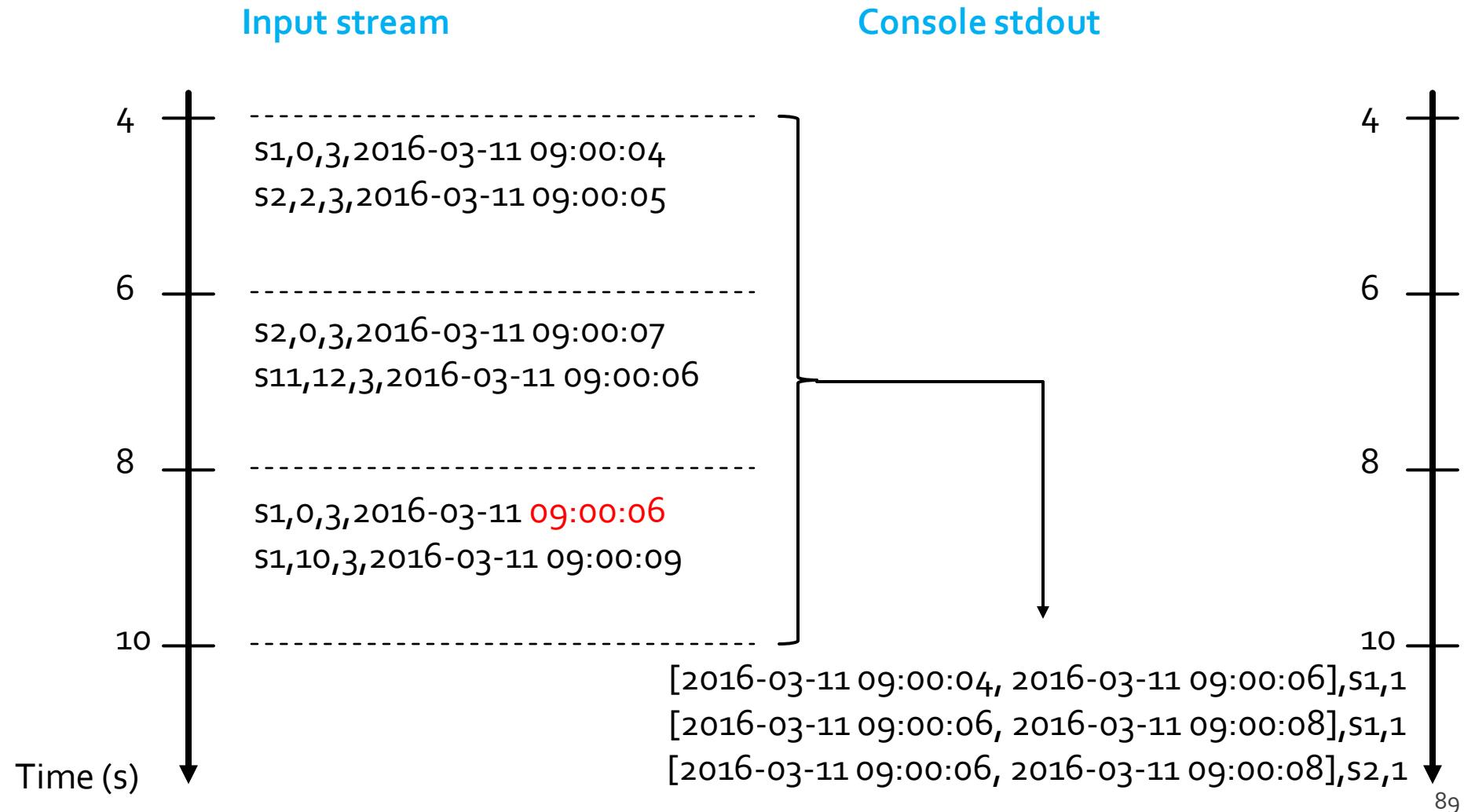
Late data: Running example



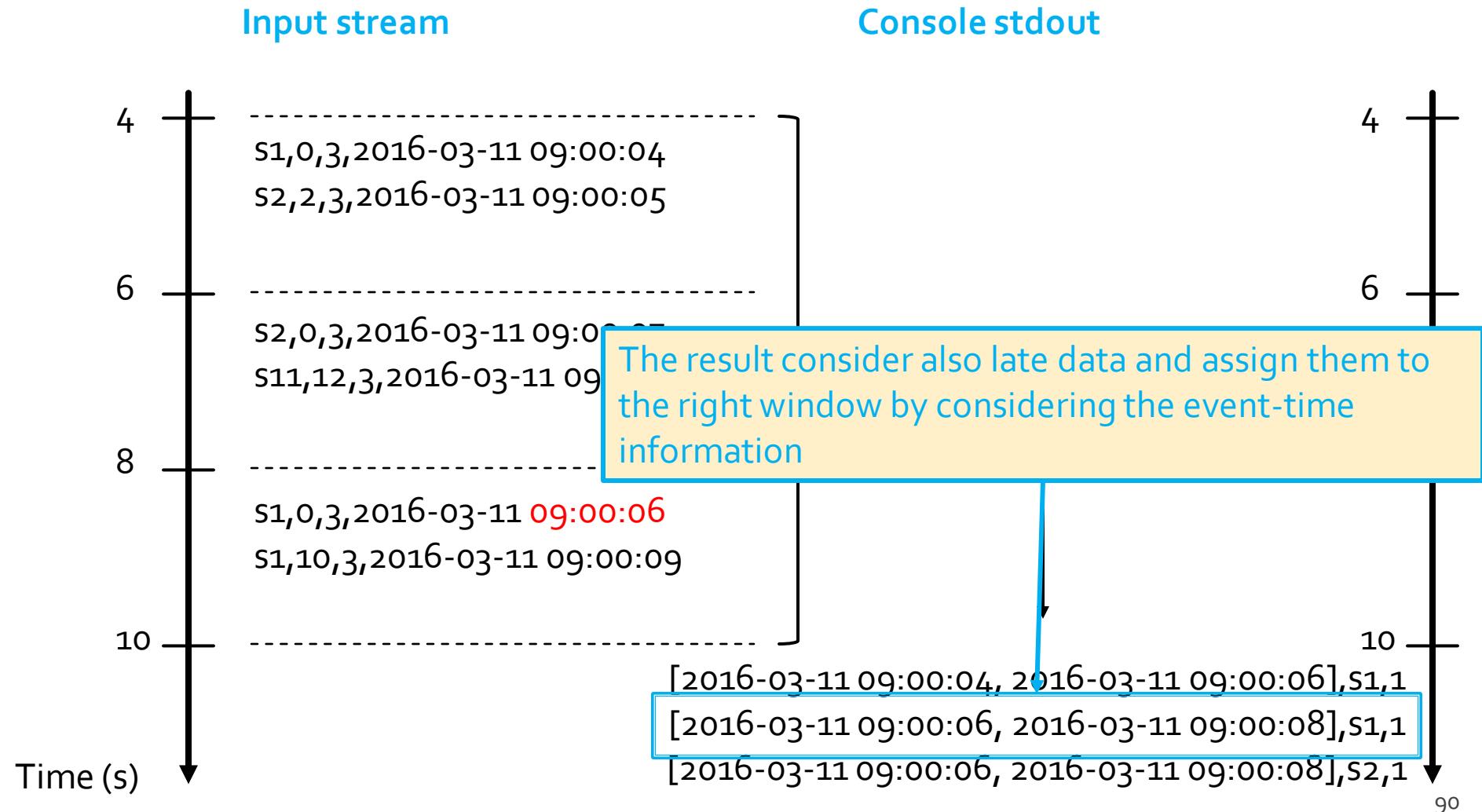
Late data: Running example



Late data: Running example



Late data: Running example



Late data: Running example

- The code is the same of “Event Time and Window Operations: Example 3”
- Late data are automatically handled by Spark

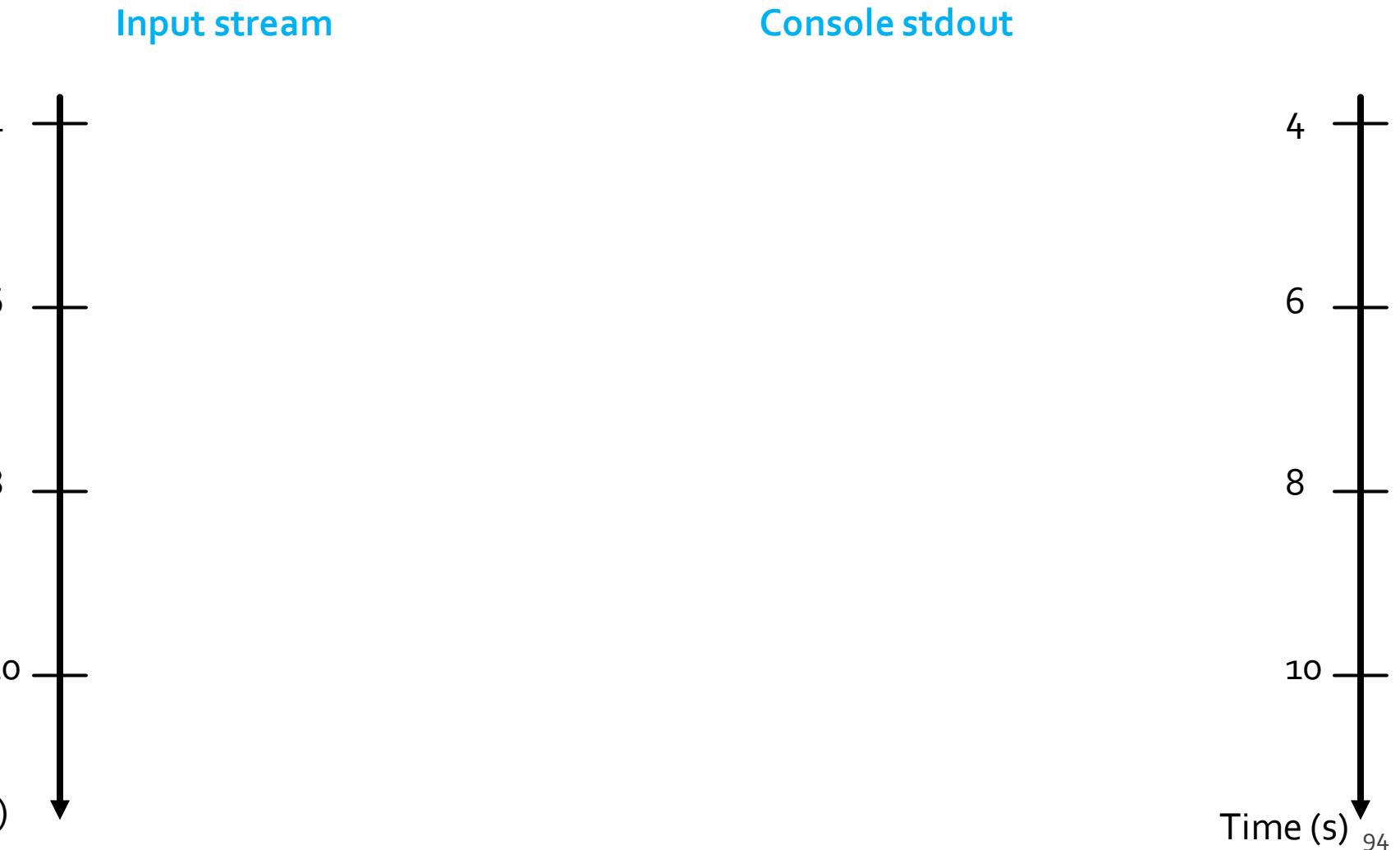
Event Time and Window Operations: Example 4

- Problem specification
 - Input
 - A stream of records retrieved from localhost:9999
 - Each input record is a reading about the status of a station of a bike sharing system in a specific timestamp
 - Each input reading has the format
 - stationId,# free slots,#used slots,timestamp
 - **timestamp** is the **event-time column**

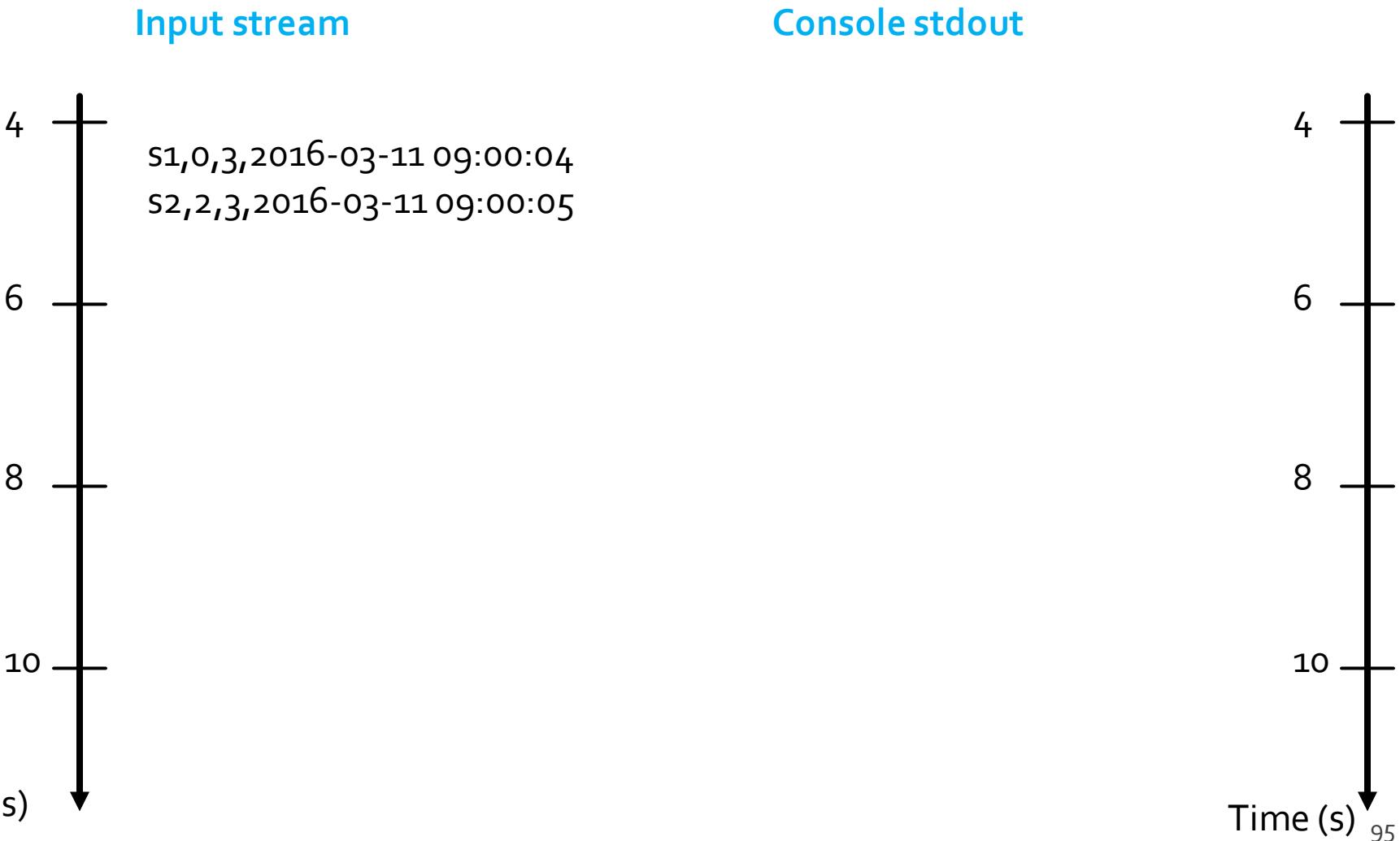
Event Time and Window Operations: Example 4

- Output
 - For each window, print on the standard output the total number of received input readings with a number of free slots equal to 0
 - The query is executed for each window
 - Set windowDuration to 2 seconds and no slideDuration
 - i.e., non-overlapped windows

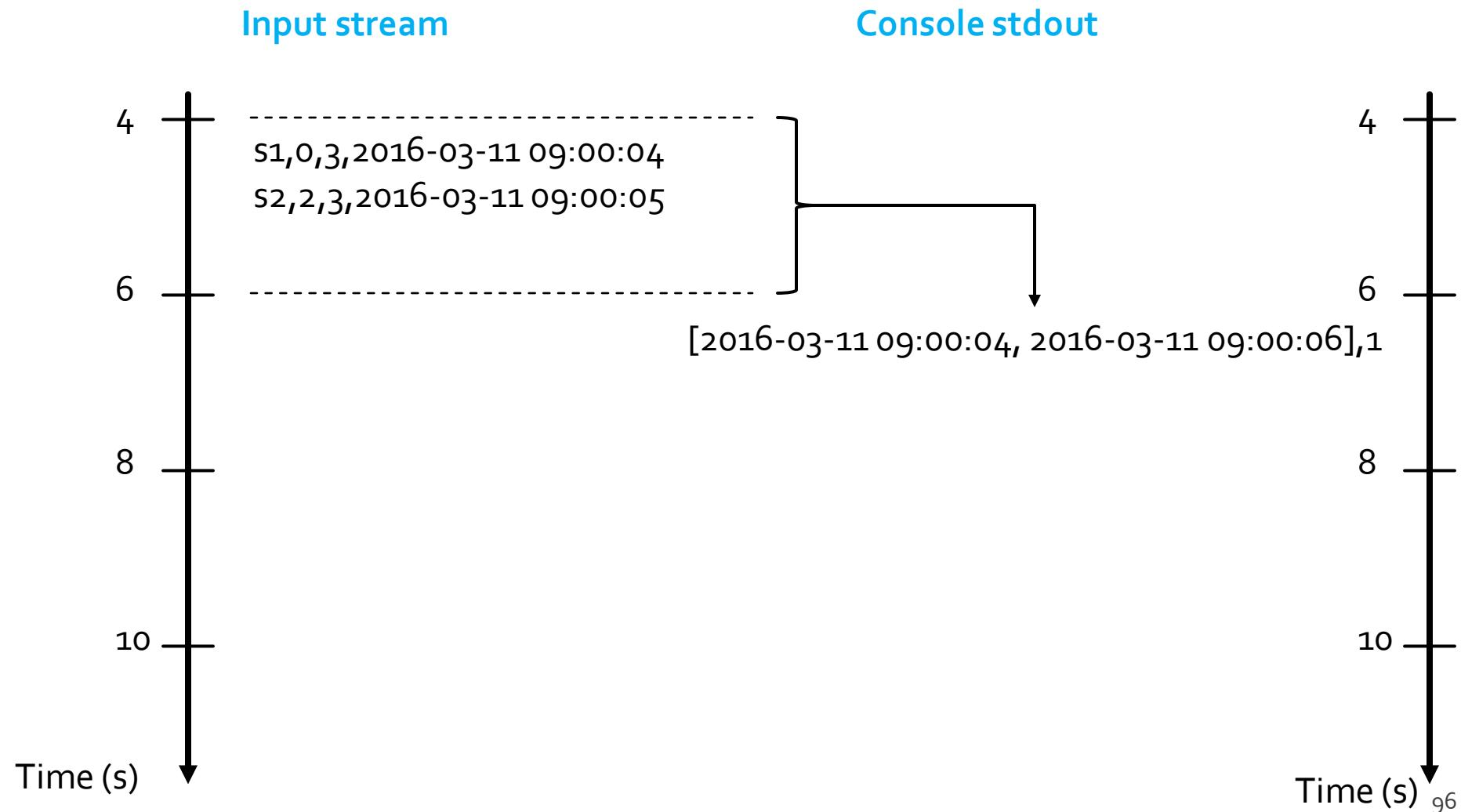
Event Time and Window Operations: Example 4



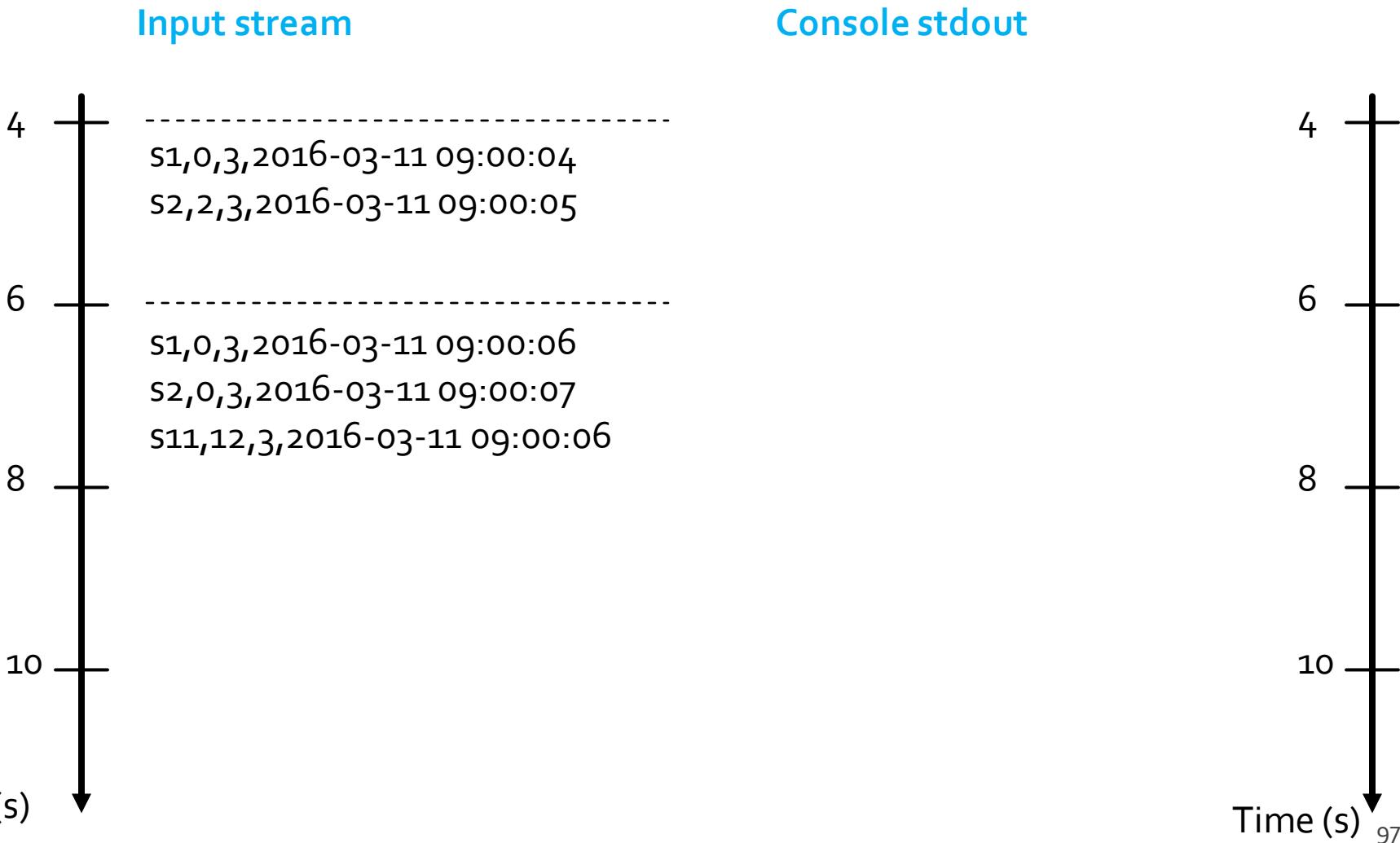
Event Time and Window Operations: Example 4



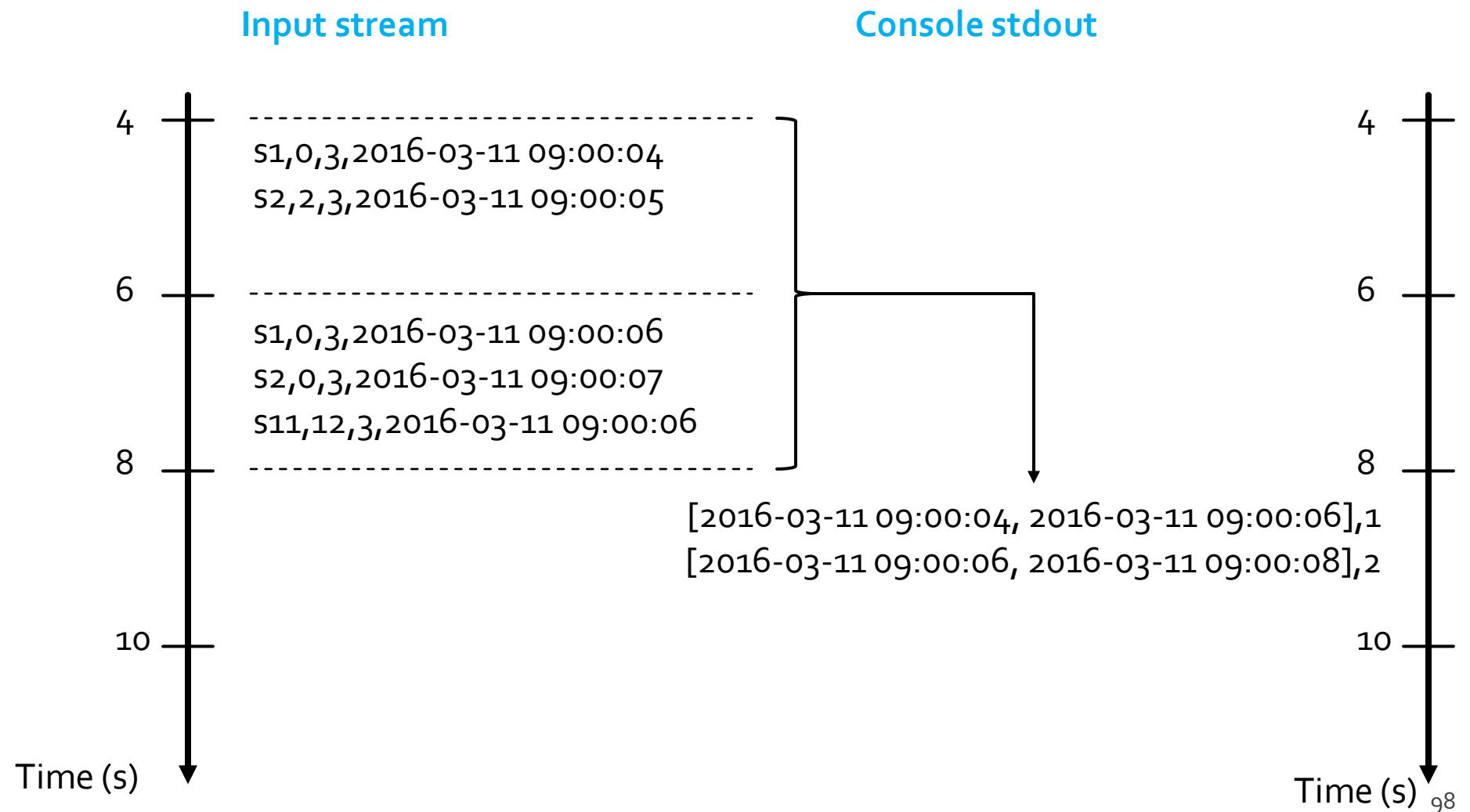
Event Time and Window Operations: Example 4



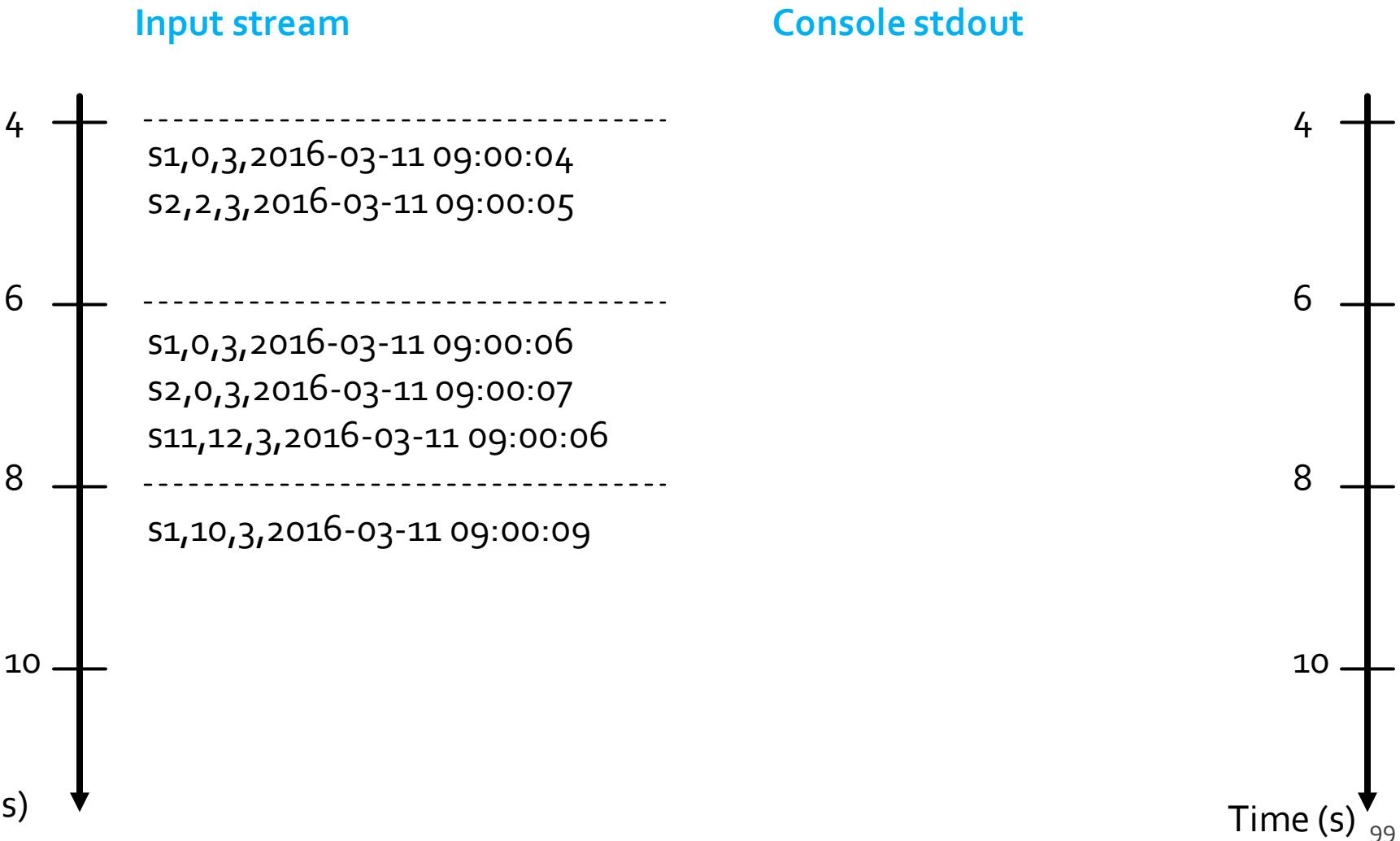
Event Time and Window Operations: Example 4



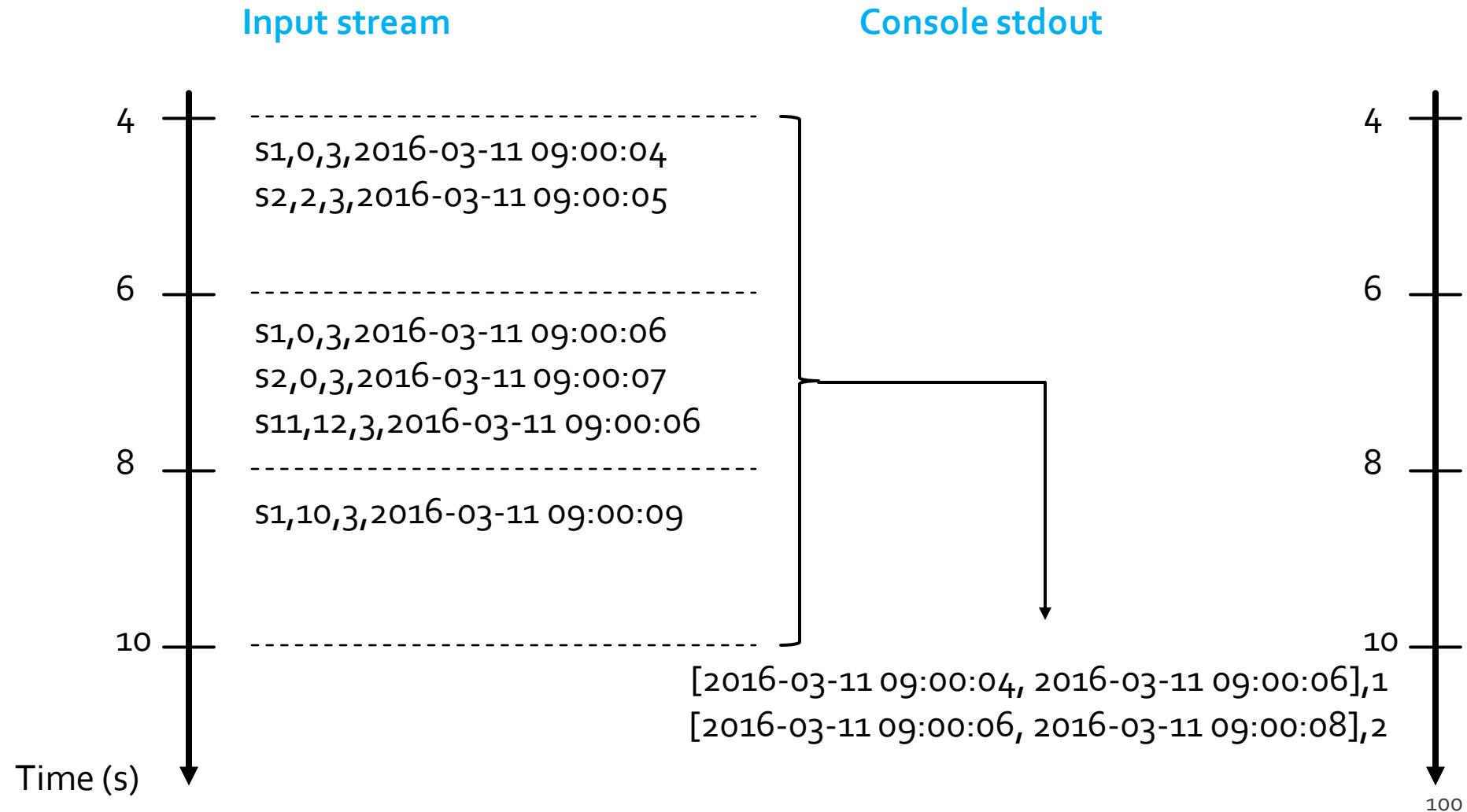
Event Time and Window Operations: Example 4



Event Time and Window Operations: Example 4



Event Time and Window Operations: Example 4



Event Time and Window Operations: Example 4

```
from pyspark.sql.types import *
from pyspark.sql.functions import split
from pyspark.sql.functions import window

# Create a "receiver" DataFrame that will connect to localhost:9999
recordsDF = spark.readStream\
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

Event Time and Window Operations: Example 4

```
# The input records are characterized by one single column called value
# of type string
# Example of an input record: s1,0,3,2016-03-11 09:00:04
# Define four more columns by splitting the input column value
# New columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF\
.withColumn("stationId", split(recordsDF.value,',')[0].cast("string"))\
.withColumn("freeslots", split(recordsDF.value,',')[1].cast("integer"))\
.withColumn("usedslots", split(recordsDF.value,',')[2].cast("integer"))\
.withColumn("timestamp", split(recordsDF.value,',')[3].cast("timestamp"))
```

Event Time and Window Operations: Example 4

```
# Filter data  
# Use the standard filter transformation  
fullReadingsDF = readingsDF.filter("freeslots=0")
```

Event Time and Window Operations: Example 4

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for in each window.
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF\
    .groupBy(window(fullReadingsDF.timestamp, "2 seconds"))\
    .agg({"*": "count"})\
    .sort("window")
```

Event Time and Window Operations: Example 4

```
# Filter data
# Use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots=0")

# Count the number of readings with a number of free slots equal to 0
# for in each window.
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF\
    .groupBy(window(fullReadingsDF.timestamp, "2 seconds"))\
    .agg({"*": "count"})\
    .sort("window")
```

We define one group for each window

Event Time and Window Operations: Example 4

```
# The result of the structured streaming query will be stored/printed on
# the console "sink"
# complete output mode
# (append mode cannot be used for aggregation queries)
queryCountWindowStreamWriter = countsDF \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .option("truncate", "false")

# Start the execution of the query (it will be executed until it is explicitly stopped)
queryCountWindow = queryCountWindowStreamWriter.start()
```

Watermarking

Watermarking

- Watermarking is a feature of Spark that allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state
- Results related to old event-times are not needed in many real streaming applications
 - They can be dropped to improve the efficiency of the application
 - Keeping the state of old results is resource expensive
- Every time new data are processed only recent records are considered

Watermarking

- Specifically, to run windowed queries for days, it is necessary for the system to bound the amount of intermediate in-memory state it accumulates
 - This means the system needs to know when an old aggregate can be dropped from the in-memory state because the application is not going to receive late data for that aggregate any more
- To enable this, in Spark 2.1, watermarking has been introduced

Watermarking

- Watermarking lets the Spark Structured Streaming engine automatically track the current event time in the data and attempt to clean up old state accordingly
- You can define the watermark of a query by specifying the event time column and the threshold on how late the data is expected to be in terms of event time
 - For a specific window ending at time T, the engine will maintain state and allow late data to update the state/the result until
$$\text{max event time seen by the engine} < T + \text{late threshold}$$
 - In other words, late data within the threshold will be aggregated, but data later than $T+\text{threshold}$ will be dropped

Join Operations

Join Operations

- Spark Structured Streaming manages also join operations
 - Between two streaming DataFrames
 - Between a streaming DataFrame and a static DataFrame
- The result of the streaming join is generated incrementally

Join Operations

- Join between two streaming DataFrames
- For both input streams, past input streaming data must be buffered/recorded in order to be able to match every future input record with past input data and accordingly generate joined results
- Too many resources are needed for storing all the input data
- Hence, **old data must be discarded**
 - You must **define watermark thresholds on both input streams** such that the engine knows how delayed the input can be and drop old data

Join Operations

- The methods `join()` and `withWatermark()` are used to join streaming DataFrames
- The join method is similar to the one available for static DataFrame

Join Operations: Example

```
from pyspark.sql.functions import expr
impressions = spark.readStream. ...
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2
    hours")

clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

# Join with event-time constraints
impressionsWithWatermark.join(
    clicksWithWatermark,
    expr("""
        clickAdId = impressionAdId AND clickTime >= impressionTime AND
        clickTime <= impressionTime + interval 1 hour
    """) )
```

Streaming data analytics: Frameworks

Stream Processing Frameworks for (Big) Streaming Data Analytics

- Several frameworks have been proposed to process in real-time or in near real-time data streams
 - Apache Spark (Streaming component)
 - Apache Storm
 - Apache Flink
 - Apache Samza
 - Apache Apex
 - Apache Flume
 - Amazon Kinesis Streams
 - ...
- All these frameworks use a cluster of servers to scale horizontally with respect to the (big) amount of data to be analyzed

Comparison among the state of the art streaming frameworks

Some of the most used Data Streaming Frameworks

- Apache Spark Streaming
 - Micro-batch applications
 - Processes each record **exactly once**

Some of the most used Data Streaming Frameworks

- Apache Storm
 - Continuous/real-time computation
 - Very low latency
 - Processes each record **at least once** in real-time
 - Each record could be processed multiple times
 - Hence, may update mutable state twice
 - Apache Storm Trident API
 - Micro-batch
 - A running “modality” of Apache Storm that processes each record **exactly once**
 - Slower than the Apache Storm version

Some of the most used Data Streaming Frameworks

- Apache Flink
 - Continuous/real-time stateful computations over data streams
 - Low latency
 - Processes each record **exactly once**

Introduction to Apache Storm

Apache Storm

- Apache Storm™ is a distributed framework that is used for real-time processing of data streams
 - E.g., Tweets analysis, Log processing, ...
- Currently, it is an open source project of the Apache Software Foundation
 - <http://storm.apache.org/>
- It is implemented in Clojure and Java
 - 12 core committers, plus ~ 70 contributors



Apache Storm history

- Storm was first developed by Nathan Marz at BackType
 - BackType was a company that provided social search applications
- Later (2011), BackType was acquired by Twitter, and now it is a critical part of their infrastructure
- Currently, Storm is a project of the Apache Software Foundation (since 2013)

Apache Storm: data processing

- Continuous computation
 - Storm can do continuous computation on data streams in real time
 - It can process each message as it comes
 - An example of continuous computation is streaming trending topics detection on Twitter
- Real-time analytics
 - Storm can analyze and extract insights or complex knowledge from data that come from several real-time data streams

Features of Storm

- Storm is
 - Distributed
 - Horizontally scalable
 - Fast
 - Fault tolerant
 - Reliable - Guaranteed data processing
 - Easy to operate
 - Programming language agnostic

Features of Storm

- Distributed
 - Storm is a distributed system than can run on a cluster of commodity servers
- Horizontally scalable
 - Storm allows adding more servers (nodes) to your Storm cluster and increase the processing capacity of your application
 - It is linearly scalable with respect to the number of nodes, which means that you can double the processing capacity by doubling the nodes

Features of Storm

- Fast
 - Storm has been reported to process up to 1 million tuples per second per node
- Fault tolerant
 - Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker (on the same node or on to another node)

Features of Storm

- Reliable - Guaranteed data processing
 - Storm provides guarantees that each message (tuple) will be processed **at least once**
 - In case of failures, Storm will replay the lost tuples
 - It can be configured to process each tuple only once
- Easy to operate
 - Storm is simple to deploy and manage
 - Once the cluster is deployed, it requires little maintenance

Features of Storm

- Programming language agnostic
 - Even though the Storm platform runs on Java Virtual Machine, the applications that run over it can be written in any programming language that can read and write to standard input and output streams

Storm core concepts

Storm

- Storm can be considered a distributed Function Programming-like processing of data streams
- It applies a set of functions, in a specific order, on the elements of the input data streams and emits new data streams
 - However, each function can store its state by means of variables
 - Hence, it is not pure functional programming

Main concepts

- Tuple
- Data Stream
- Spout
- Bolt
- Topology

Data model

- The basic unit of data that can be processed by a Storm application is called a **tuple**
- Each tuple is a predefined list of fields
 - The data type of each field can be common data types, e.g., byte, char, string, integer, ..
 - Or your own data types, which can be serialized as fields in a tuple
- Each field of a tuple has a name

Data model

- A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their data type

Data model

- Storm processes streams of tuples
 - Each stream is an unbounded sequence of tuples
- Each stream
 - has a name
 - is composed of homogenous tuples (i.e., tuples with the same structure)
- However, each application can process multiple, heterogeneous, data streams

Data model: Example

- Tuple
 - (1.1.1.1, "foo.com")
IP address Domain
- Stream of tuples
 - ...
 - (1.1.1.1, "foo.com")
 - (2.2.2.2, "bar.net")
 - (3.3.3.3, "foo.com")
 - ...

Spout

- Spout
 - It is the component “generating/handling” the input data stream
- Spouts read or listen to data from external sources and publish them (emit in Storm terminology) into streams

Spout

- Examples
 - A spout can be used to connect to the Twitter API and emit a stream of tweets
 - A spout can be used to read a log file and emit a stream of composed of the its lines
 - ...

Spout

- Each spout can emit multiple streams, with different schemas
 - For example, we can implement a spout that reads 10-field records from a log file and emits them as two different streams of 7-tuples and 4-tuples, respectively
- Spouts can be
 - “unreliable” (fire-and-forget)
 - or “reliable” (can replay failed tuples)

Bolt

- Bolt
 - It is the component that is used to apply a function over each tuple of a stream
- Bolts consume one or more streams, emitted by spouts or other bolts, and potentially produce new streams

Bolt

- Bolts can be used to
 - Filter or transform the content of the input streams and emit new data streams that will be processed by other bolts
 - Or process the data streams and store/persist the result of the computation in some of “storage” (files, Databases, ..)
- Each bolt can emit multiple streams, with different schemas

Bolt

- Examples
 - A bolt can be used to extract one field from each tuple of its input stream
 - A bolt can be used to join two streams, based on a common field
 - A bolt can be used to count the occurrences of a set of URLs
 - ...

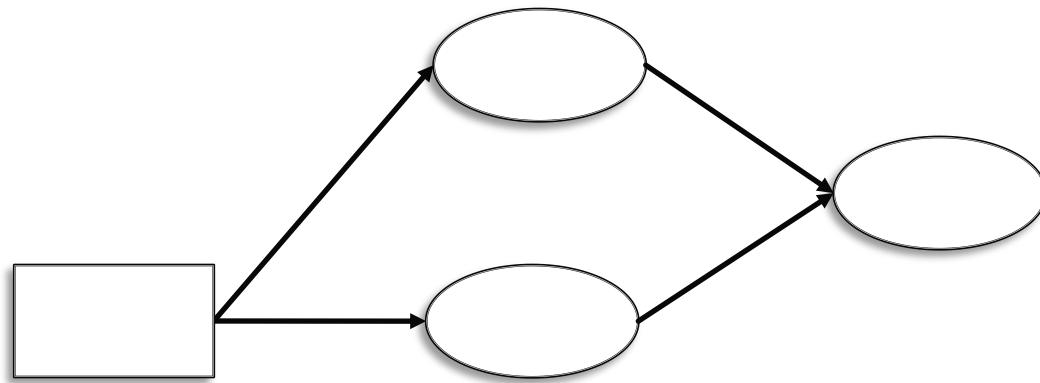
Spouts and Bolts

- The input streams of a Storm cluster are handled by spouts
- Each spout passes the data streams to bolts, which transform them in some way
- Each bolt either persists the data in some sort of storage or passes it to some other bolts
- A Storm program is a chain of bolts making some computations/transformations on the data exposed by spouts and bolts

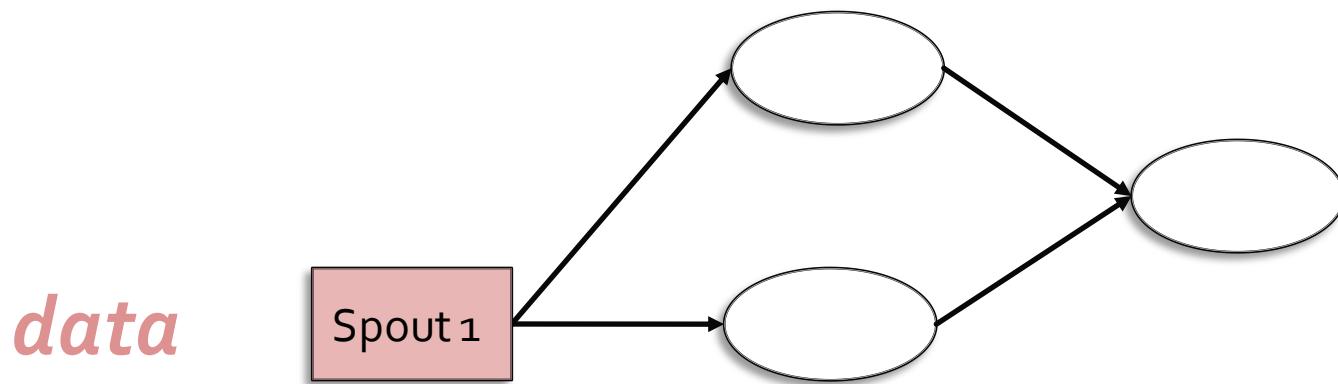
Topology

- A Storm topology is an abstraction that defines the graph of the computation
 - It specifies which spouts and bolts are used and how they are connected
- A topology can be represented by a direct acyclic graph (DAG), where each node does some kind of processing and eventually forwards it to the next node(s) in the flow
 - i.e., a topology in Storm wires **data and functions** via a **DAG**

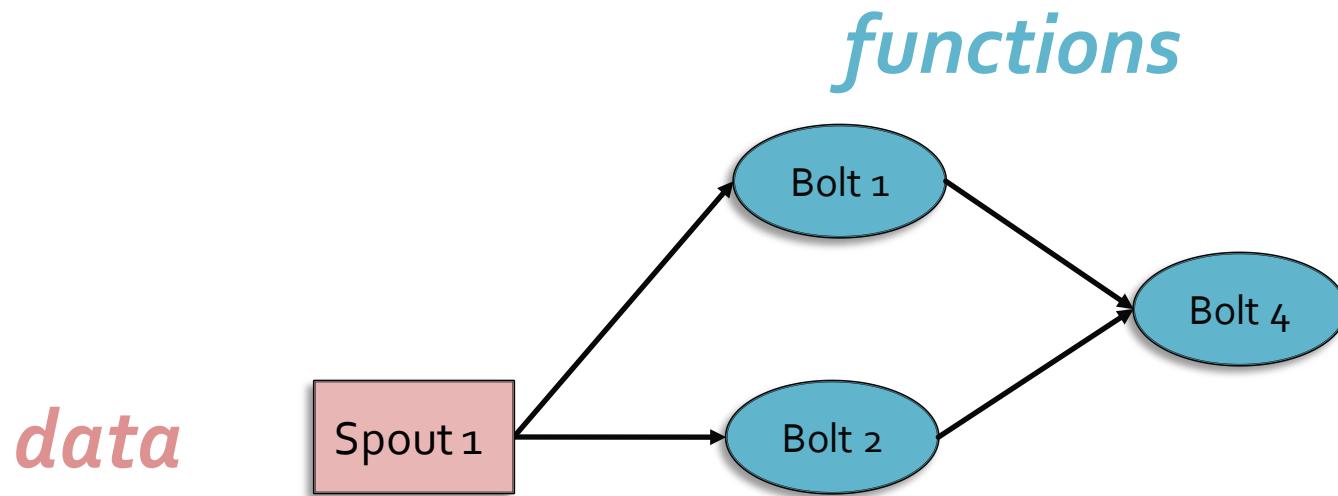
Topology: Example



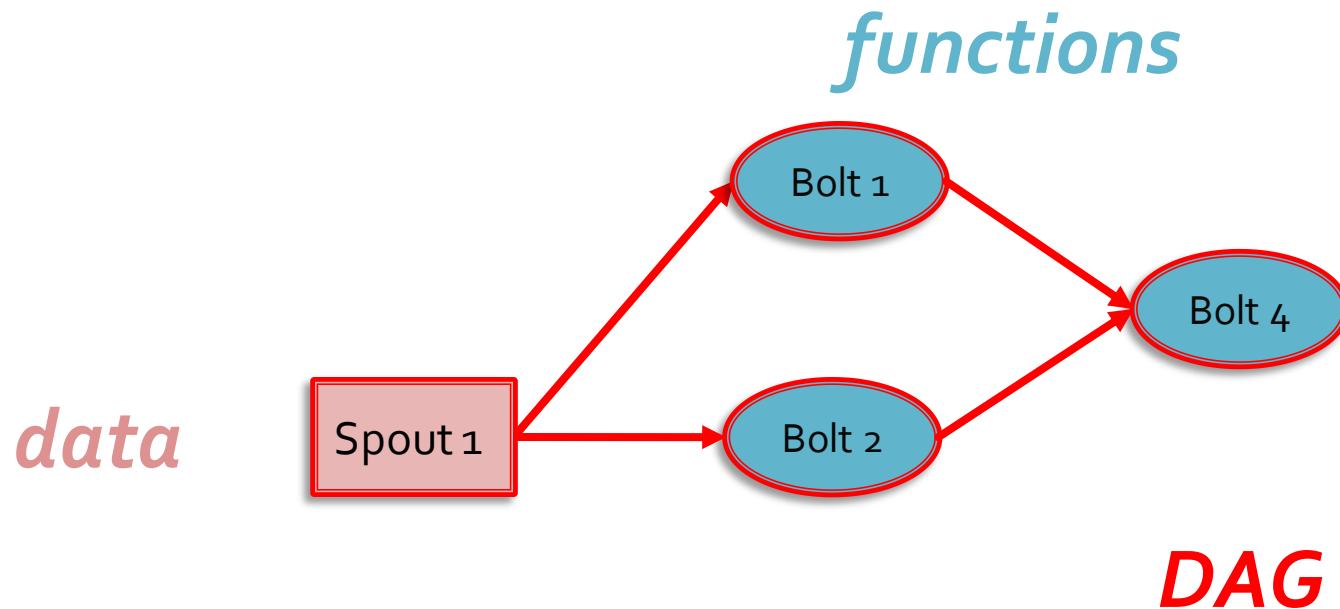
Topology: Example



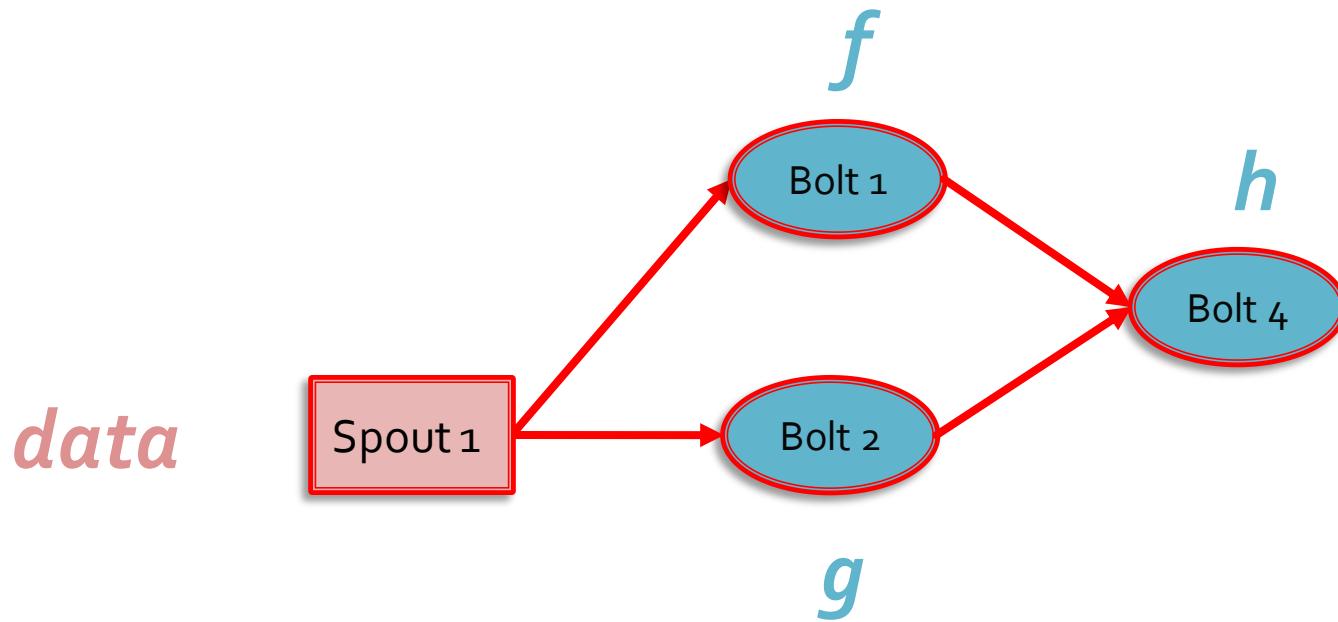
Topology: Example



Topology: Example

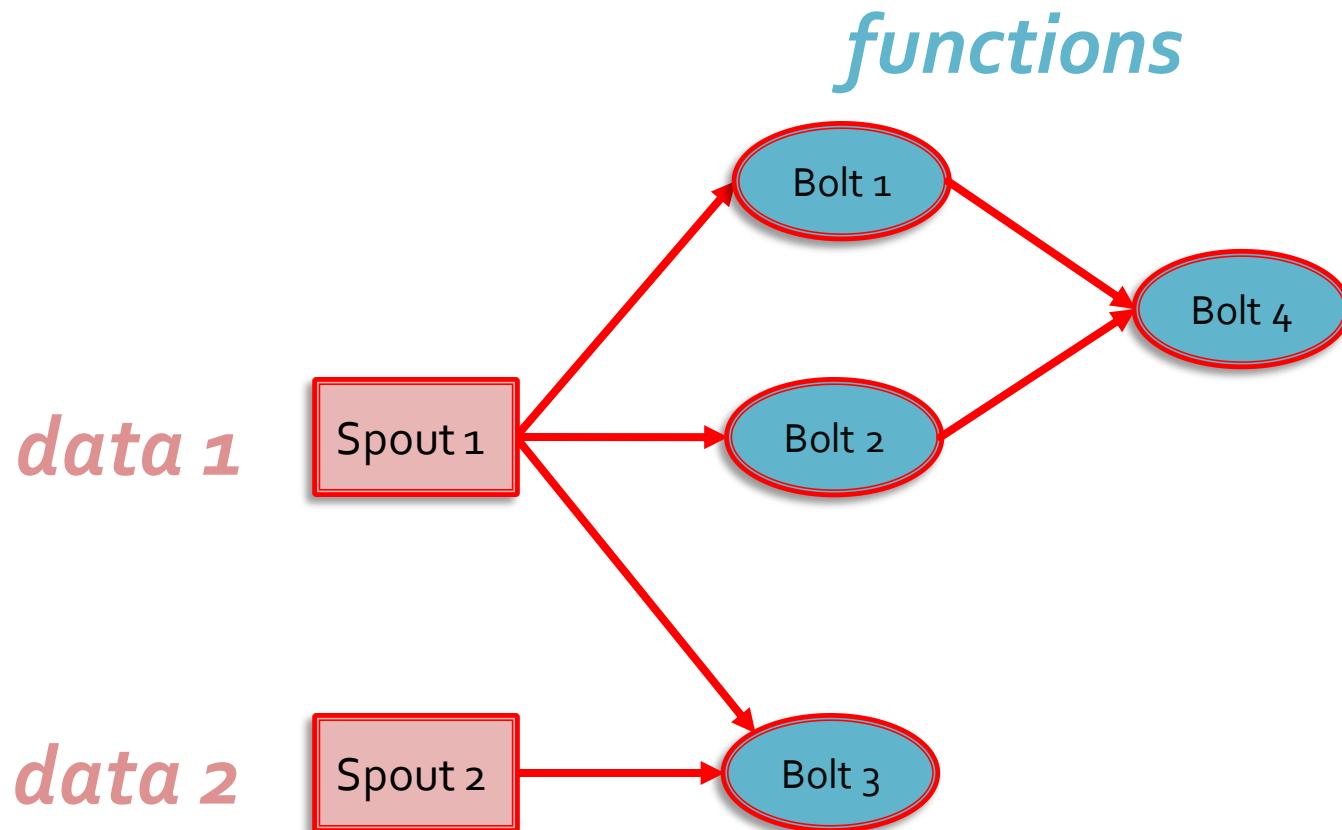


Relation Topology – Functional programming



$h(f(data), g(data))$

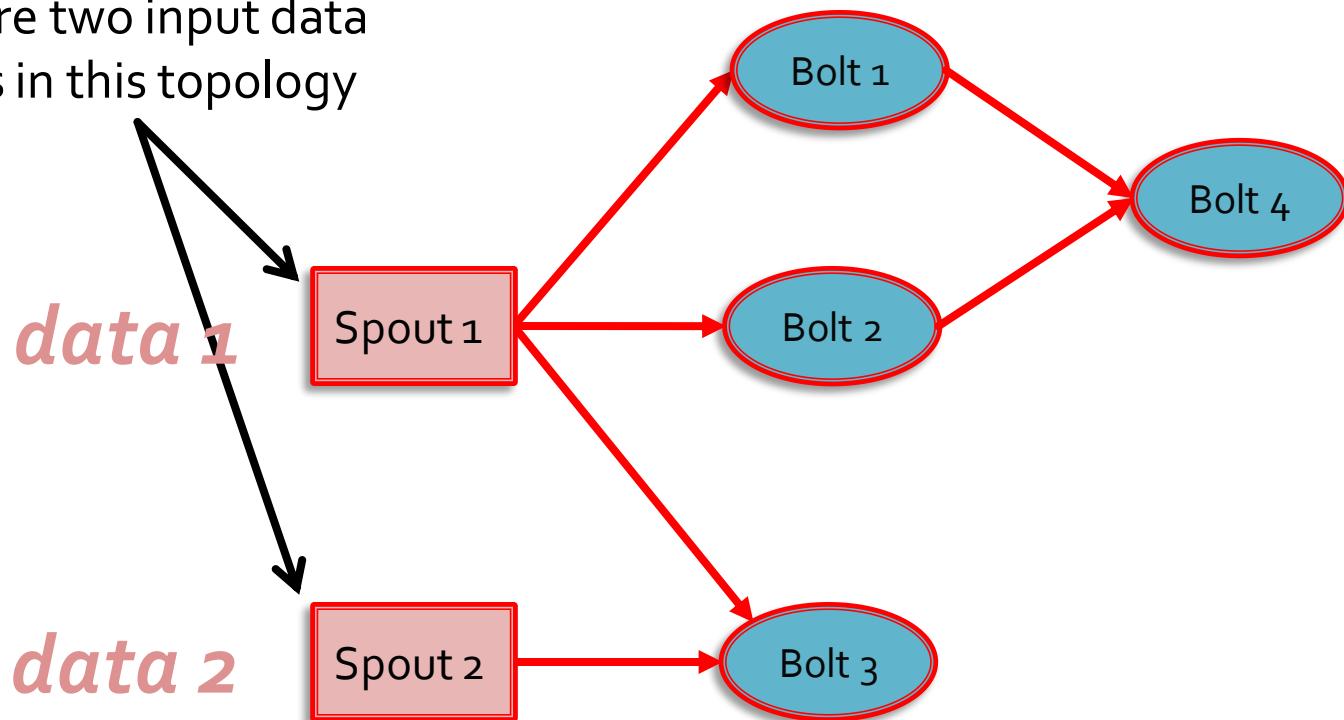
Topology: Example #2



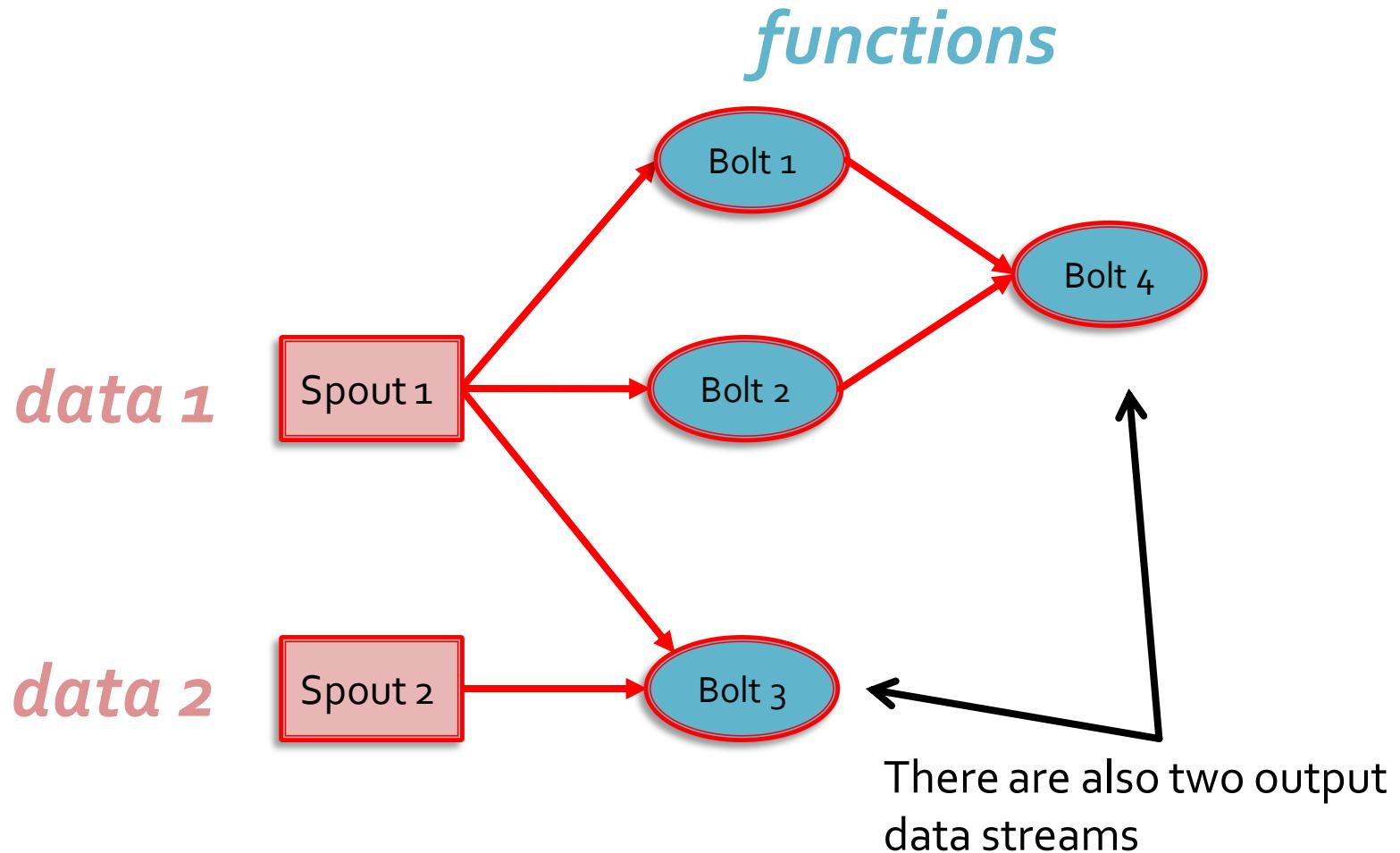
Topology: Example #2

functions

There are two input data streams in this topology



Topology: Example #2



“Execution” of a Topology

- The topology is executed on the servers of the cluster running Storm
 - The system automatically decides which parts of the topology are executed by each server of the cluster
- Each topology runs until its is explicitly killed
- Each cluster can runs multiple topologies at the same time

“Execution” of a Topology

- Worker processes
 - Each node in the cluster can run one or more JVMs called **worker processes** that are responsible for processing a part of the topology.
 - Each topology executes across one or more worker processes
 - Each worker process is bound to one of the topologies and can execute multiple components (spouts and/or bolts) of that topology
 - Hence, even if multiple topologies are run at the same time, none of them will share any of the workers

“Execution” of a Topology

- Executor
 - Within each worker process, there can be multiple threads that execute parts of the topology. Each of these threads is called an **executor**
 - An executor can execute only one of the components of the topology, that is, any one spout or bolt in the topology
 - But it may run one or more tasks for the same component
 - Each spout or bolt can be associated with many executors and hence executed in parallel

“Execution” of a Topology

- Tasks
 - A task is the most granular unit of task execution in Storm
 - Each task is an instance of a spout or bolt
 - A task performs the actual data processing
 - Each spout or bolt that you implement in your code executes as many tasks across the cluster
 - Each task can be executed alone or with another task of the same type (in the same executor)

“Execution” of a Topology

- The number of tasks for a component is always the same throughout the lifetime of a topology
 - You set it when you submit the topology
- But the number of executors (threads) for a component can change over time
 - You can add/remove executors for each component

Parallelism of the topology

- The parallelism of the topology is given by the number of executors = number of threads
- For each spout/bolt the application can specify
 - The number of executors
 - This value can be changed at runtime
 - The number of tasks
 - This value is set before submitting the topology and cannot be change at runtime