Single-pass Graph Stream Analytics with Apache Flink

Vasia Kalavri <vasia@apache.org>
Paris Carbone <senorcarbone@apache.org>
Outline

• Why Graph Streaming?
• Single-Pass Algorithms Examples
• Apache Flink Streaming API
• The GellyStream API
Real Graphs are *dynamic*

Graphs are created from **events** happening in real-time
Vasia Kalavri @vkalavri · 9 Dec 2015
Just submitted a talk w/ @SenorCarbone at the FOSDEM @GraphDevroom! Have you submitted yours? CfP closes Dec 14 graphdevroom.github.io

GraphDevroom Retweeted
Vasia Kalavri @vkalavri · 9 Dec 2015
Just submitted a talk w/ @SenorCarbone at the FOSDEM @GraphDevroom! Have you submitted yours? CfP closes Dec 14 graphdevroom.github.io

Christophe Willemsen @ikwattro · 9 Dec 2015
@vkalavri @SenorCarbone @GraphDevroom looking forward to your talk !!

Paris Carbone Retweeted
Vasia Kalavri @vkalavri · 9 Dec 2015
Just submitted a talk w/ @SenorCarbone at the FOSDEM @GraphDevroom! Have you submitted yours? CfP closes Dec 14 graphdevroom.github.io
Batch Graph Processing

We create and analyze a **snapshot** of the real graph

- the Facebook social network on January 30 2016
- user web logs gathered between March 1st 12:00 and 16:00
- retweets and replies for 24h after the announcement of the death of David Bowie
Streaming Graph Processing

We consume events in **real-time**

- Get results *faster*
  - No need to wait for the job to finish
  - Sometimes, early approximations are better than late exact answers

- Get results *continuously*
  - Process *unbounded* number of events
Challenges

• Maintain the graph structure
  • How to apply state updates efficiently?

• Result updates
  • Re-run the analysis for each event?
  • Design an incremental algorithm?
  • Run separate instances on multiple snapshots?

• Computation on most recent events only
Single-Pass Graph Streaming

- Each event is an edge addition
- Maintains only a graph summary
- Recent events are grouped in graph windows
Streaming Degrees Distribution

![Graph Diagram]

- Degree distribution

#vertices

<table>
<thead>
<tr>
<th>degree</th>
<th>#vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

@GraphDevroom
Streaming Degrees Distribution

![Graph Diagram]

#vertices

degree
Streaming Degrees Distribution

![Graph Diagram]

degree

#vertices

0 1 2 3 4 5 6

1 2 3 4 5 6 7 8
Streaming Degrees Distribution

#vertices

degree
Streaming Degrees Distribution

![Diagram of a graph with labeled vertices and a bar chart showing the degree distribution](image)
Streaming Degrees Distribution

![Graph Diagram]

1. Degree Distribution

- Degree 1: 1 vertex
- Degree 2: 2 vertices
- Degree 3: 5 vertices
- Degree 4: 8 vertices

@GraphDevroom
Streaming Degrees Distribution

#vertices | degree
---|---
1 | 5
2 | 3
3 | 1
4 | 2
5 | 2
6 | 1
7 | 2
8 | 1
Streaming Degrees Distribution

#vertices

degree

#vertices

0 1 2 3 4 5 6

1 2 3 4
Streaming Degrees Distribution

![Graph with vertices and degrees]

- Vertices: 1, 2, 3, 4, 5, 6, 7, 8
- Degrees: 1, 4, 3, 2, 5, 6, 7, 8
Streaming Degrees Distribution

![Graph Diagram]

- #vertices
- degree

Histogram showing the distribution of vertices by degree.

- Degree 1: 1 vertex
- Degree 2: 3 vertices
- Degree 3: 2 vertices
- Degree 4: 1 vertex

@GraphDevroom
Graph Summaries

- **spanners** for distance estimation
- **sparsifiers** for cut estimation
- **sketches** for homomorphic properties
Window Aggregations

Neighborhood aggregations on windows
Examples
Batch Connected Components

- **State:** the graph and a component ID per vertex (initially equal to vertex ID)

- **Iterative **Computation:** For each vertex:
  
  - choose the min of neighbors’ component IDs and own component ID as new ID
  
  - if component ID changed since last iteration, notify neighbors
Batch Connected Components

\[ i=0 \]

Graph: 

- Nodes: 1, 2, 3, 4, 5, 6, 7, 8
- Connections: 
  - 1 to 2
  - 1 to 3
  - 2 to 4
  - 4 to 5
  - 6 to 7
  - 8 to 7

Note: The graph is divided into components.
Batch Connected Components

i=1

![Graph Diagram]
Batch Connected Components

\[ i=2 \]
Batch Connected Components

\[ i = 3 \]
Stream Connected Components

- **State:** a *disjoint set* data structure for the components

- **Computation:** For each edge
  - if seen for the 1st time, create a component with ID the min of the vertex IDs
  - if in different components, *merge* them and update the component ID to the min of the component IDs
  - if only one of the endpoints belongs to a component, add the other one to the same component
@ComponentID   | Vertices
---|---

1
2
3
4
5
6
7
8
ComponentID | Vertices
---|---
1 | 1, 3
2 | 2, 4, 5
<table>
<thead>
<tr>
<th>ComponentID</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3</td>
</tr>
<tr>
<td>2</td>
<td>2, 4, 5</td>
</tr>
<tr>
<td>6</td>
<td>6, 7, 8</td>
</tr>
</tbody>
</table>

![Graph Diagram]
ComponentID | Vertices
--- | ---
1 | 1, 3
2 | 2, 4, 5
6 | 6, 7, 8
<table>
<thead>
<tr>
<th>ComponentID</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3</td>
</tr>
<tr>
<td>2</td>
<td>2, 4, 5</td>
</tr>
<tr>
<td>6</td>
<td>6, 7, 8</td>
</tr>
</tbody>
</table>

Diagram: A graph with nodes 1, 2, 3, 4, 5, 6, 7, 8 connected as follows:
- 1 is connected to 2 and 4.
- 2 is connected to 4 and 5.
- 3 is connected to 4.
- 5 is connected to 4.
- 6 is connected to 7.
- 7 is connected to 8.
ComponentID | Vertices
---|---
1 | 1, 2, 3, 4, 5
6 | 6, 7, 8
### ComponentID vs Vertices

<table>
<thead>
<tr>
<th>ComponentID</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>6</td>
<td>6, 7, 8</td>
</tr>
</tbody>
</table>
### ComponentID vs Vertices

<table>
<thead>
<tr>
<th>ComponentID</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>6</td>
<td>6, 7, 8</td>
</tr>
</tbody>
</table>
Distributed Stream Connected Components
Stream Bipartite Detection

Similar to connected components, but

• Each vertex is also assigned a sign, (+) or (-)

• Edge endpoints must have different signs

• When merging components, if flipping all signs doesn’t work => the graph is not bipartite
Stream Bipartite Detection

Cid=1

Cid=5

(+) 1 2 (-)
(+) 3 4 (-)
(+ ) 5 7 (-)
(+ ) 6
Stream Bipartite Detection

Cid=1

Cid=5
Stream Bipartite Detection

Cid=1

Cid=5
Stream Bipartite Detection

Cid=1

(+) 1 2 (-)
(+) 3 4 (-)
(+) 7 5 (-)

3 5

6 (-)
Stream Bipartite Detection

Can’t flip signs and stay consistent
=> not bipartite!
API Requirements

• Continuous aggregations on edge streams
• Global graph aggregations
• Support for windowing
The Apache Flink Stack

- Bounded Data Sources
- Blocking Operations
- Structured Iterations

- Unbounded Data Sources
- Continuous Operations
- Asynchronous Iterations

APIs

Execution

DataSet

DataStream

Distributed Dataflow

Deployment
Unifying Data Processing

**Client**
- execution plan building
- optimisation

**Job Manager**
- scheduling tasks
- monitoring/recovery

**DataStream**
- task pipelining
- blocking

**Dataset**
- text = env.readTextFile("hdfs://...");
  text.map(...).groupReduce(...)

**Dataset**
- text = env.readTextFile("hdfs://...");
  text.map(...).groupReduce(...)

**Deployment**

**Distributed Dataflow**

**HDFS**

**Kafka**
Data Streams as ADTs

- **Transformations**: map, flatmap, filter, union…
- **Aggregations**: reduce, fold, sum
- **Partitioning**: forward, broadcast, shuffle, keyBy
- **Sources/Sinks**: custom or Kafka, Twitter, Collections…

- **Tasks** are long running in a pipelined execution.
- **State** is kept within tasks.
- **Transformations** are applied per-record or window.
Working with Windows

Why windows?
We are often interested in fresh data!

Highlight: Flink can form and trigger windows consistently under different notions of time and deal with late events!

1) Sliding windows
   ```java
   myKeyStream.timeWindow(
       Time.of(60, TimeUnit.SECONDS),
       Time.of(20, TimeUnit.SECONDS));
   ```

2) Tumbling windows
   ```java
   myKeyStream.timeWindow(
       Time.of(60, TimeUnit.SECONDS));
   ```
Example

```java
myTextStream
    .flatMap(new Splitter())  //transformation
    .keyBy(0)  //partitioning
    .window(Time.of(5, TimeUnit.MINUTES))
    .sum(1)  //rolling aggregation
    .setParallelism(4);

counts.print();
```

11:01 - “dataflow is cool too”
10:48 - “cool, gelly is cool”
Gelly on Streams

- Static Graphs
- Multi-Pass Algorithms
- Full Computations

- Dynamic Graphs
- Single-Pass Algorithms
- Approximate Computations
Introducing Gelly-Stream

**Gelly-Stream** enriches the DataStream API with two new additional ADTs:

- **GraphStream**:
  - A representation of a data *stream of edges*.
  - Edges can have *state* (e.g. weights).
  - Supports *property* streams, *transformations* and *aggregations*.

- **GraphWindow**:
  - A “time-slice” of a graph stream.
  - It enables neighbourhood aggregations
GraphStream Operations

<table>
<thead>
<tr>
<th>Property Streams</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GraphStream -&gt; DataStream</strong></td>
<td><strong>GraphStream -&gt; GraphStream</strong></td>
</tr>
<tr>
<td>.getEdges()</td>
<td>.mapEdges();</td>
</tr>
<tr>
<td>.getVertices()</td>
<td>.distinct();</td>
</tr>
<tr>
<td>.numberOfVertices()</td>
<td>.filterVertices();</td>
</tr>
<tr>
<td>.numberOfEdges()</td>
<td>.filterEdges();</td>
</tr>
<tr>
<td>.getDegrees()</td>
<td>.reverse();</td>
</tr>
<tr>
<td>.inDegrees()</td>
<td>.undirected();</td>
</tr>
<tr>
<td>.outDegrees()</td>
<td>.union();</td>
</tr>
</tbody>
</table>
Graph Stream Aggregations

```java
graphStream.aggregate(
    new MyGraphAggregation(window, fold, combine, transform))
```

- **fold**: (window) fold
- **reduce combine**: global aggregates can be **persistent** or **transient**
- **result property stream**:
- **local summaries**
- **global summary**

GraphStream.

- **graph stream**
- **edges**

@GraphDevroom
Graph Stream Aggregations

```java
graphStream.aggregate(
    new MyGraphAggregation(window, fold, combine, transform)
)
```
Connected Components

graphStream.aggregate(
new ConnectedComponents(window,fold,combine,transform))

#components
graphStream.aggregate(
    new ConnectedComponents(window, fold, combine, transform))

#components
Connected Components

```java
graphStream.aggregate(
    new ConnectedComponents(window, fold, combine, transform))
```

Diagram showing the connected components of a graph stream.
Connected Components

graphStream.aggregate(
new ConnectedComponents(window, fold, combine, transform))

#components
Connected Components

graphStream.aggregate(
new ConnectedComponents(window,fold,combine,transform))

graphStream

{6,7}
{1,3}
{4,5}

window
triggers

{2,5}
{6,8}

#components
Connected Components

graphStream.aggregate(
new ConnectedComponents(window, fold, combine, transform))

#components
Connected Components

graphStream.aggregate(
new ConnectedComponents(window, fold, combine, transform))

#components
Connected Components

```java
graphStream.aggregate(
    new ConnectedComponents(window, fold, combine, transform))
```

#components
Connected Components

graphStream.aggregate(
new ConnectedComponents(window, fold, combine, transform))
Connected Components

```java
graphStream.aggregate(
    new ConnectedComponents(window, fold, combine, transform))
```

The graph stream is processed through a `ConnectedComponents` operation. This operation takes a window, fold, combine, and transform function to identify and aggregate connected components in the graph stream. The output is the number of components.

The diagram illustrates the flow of the graph stream and the aggregation of connected components.
Connected Components

```java
graphStream.aggregate(new ConnectedComponents(window, fold, combine, transform))
```

#components
graphStream.aggregate(new ConnectedComponents(window, fold, combine, transform))
Aggregating Slices

```java
graphStream.slice(Time.of(1, MINUTE), direction)
```

- Slicing collocates edges by vertex information
- Neighbourhood aggregations are now enabled on sliced graphs
Finding Matches Nearby

graphStream.filterVertices(GraphGeeks()).
slice(Time.of(15, MINUTE), EdgeDirection.IN).
applyOnNeighbors(FindPairs())
Feeling Gelly?

- **Gelly-Stream**: https://github.com/vasia/gelly-streaming

- **Apache Flink**: https://github.com/apache/flink

- **An interesting read**: http://users.dcc.uchile.cl/~pbarcelo/mcg.pdf

- **A cool thesis**: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-170425

- **Twitter**: @vkalavri, @senorcarbone