
DEGREE PROJECT, IN DISTRIBUTED SYSTEMS , SECOND LEVEL
*STOCKHOLM*, *SWEDEN 2015*

# Streaming Graph Analytics Framework Design

## DEGREE PROJECT IN DISTRIBUTED COMPUTING AT KTH INFORMATION AND COMMUNICATION TECHNOLOGY

JÁNOS DÁNIEL BALI

**KTH ROYAL INSTITUTE OF TECHNOLOGY**

**KTH INFORMATION AND COMMUNICATION TECHNOLOGY**

TRITA TRITA-ICT-EX-2015:169


KTH Royal Institute of Technology

School of Information and Communication Technology

Degree project in Distributed Computing

# Streaming Graph Analytics Framework Design

Author: János Dániel Bali
Supervisors: Vasiliki Kalavri
Paris Carbone

Examiner: Vladimir Vassov, KTH, Sweden

**Abstract**

Along with the spread of the World Wide Web, social networks and the Internet of Things in the past decades, the need for systems and algorithms that can process massive graphs has been continuously increasing. There has been considerable amount of research done in distributed Graph processing since the emergence of such large-scale graphs.

Another steadily growing field in the past years has been stream processing. This rise of interest can be attributed to the need to process large amounts of continuously streaming data with scalability, fault tolerance and very low latency.

Graph streaming, the unification of those two fields is a rather new idea, with some research already being done on it. Processing graphs that are unbounded, and so large that they cannot be stored in memory or even on the disk, is only possible with a distributed graph streaming model.

Our goal is to provide a graph streaming model and API that can handle common transformations and provide statistics on streamed graphs. This graph streaming API is created on top of Flink streaming and provides similar interfaces to Gelly, which is the graph library on the batch processing part of Flink.

## Referat

Spridningen av World Wide Web, sociala nätverk och Internet of Things under de senaste decennierna har behovet av system och algoritmer som kan bearbeta stora grafer kontinuerligt ökat. Det har skett en betydande mångd olika forskningar i distribuerade graf bearbetningar på grund av uppkomsten av sådana storskaliga grafer.

Stream-proccesing har under de senaste åren varit ett stadigt växande område. Denna ökning av intresse kan hänföras till behovet av att bearbeta stora mängder av kontinuerligt strömmande data med skalbarhet, feltolerans och mycket låg latens.

Graph streaming, enandet av dessa två områden är en ganska ny idé, med en del efterforskningar som redan görs idag. Bearbetning av grafer som är obegränsade, och så stora att de inte kan lagras i ett minne eller på en hårddisk, är bara möjligt med en distribuerad graf streaming modell.

Vårt mål är att ge en Graph Streaming Model och API som kan hantera gemensamma transformationer och ta fram statistik på en streamad graf. Denna typ av graf streaming API skapas ovanpå Flink streaming och tillhandahåller liknande gränssnitt till Gelly, som är graf biblioteket på batch-bearbetnings delen av Flink.

# Acknowledgment

I would like to express my deepest gratitude to my supervisors, Vasia Kalavri and Paris Carbone for their continuous support and encouragement. Working with them has been a great experience and pleasure. I would also like to thank Martha Vlachou, Faye Beligianni and Gyula Fóra for their help and friendliness during the time I was working on my degree project at the Swedish Institute of Computer Science.

Stockholm, June 30, 2015

*János Dániel Bali*

# Contents

# List of Figures

# List of Tables

# Chapter 1
# 1 Introduction

## 1.1 Motivation

The need to process gigantic graphs with a very high throughput has been ever-increasing in the past years. Several large-scale graph processing engines have gained popularity in the past years, including Apache Giraph(6), an open-source implementation based on Google's Pregel(13). These engines offer very good performance when working on large, but static or bounded graphs. They support different computation strategies, such as the vertex-centric(13) and the gather-apply-scatter models(11).

An alternative to the traditional batch processing approach is data streaming. The main idea behind streaming is to be able to process very large amounts of data without having enough storage for the whole data, while still maintaining very low latency. Also, with stream processing, some results can be obtained before the whole data is processed. An example of this is bipartiteness — as soon as we find a counter-example to the graph being bipartite, we can terminate the entire algorithm. Stream processing is also immensely useful when computing long running statistics.

Many different data processing frameworks have been released in the past years, including the Apache project's Hadoop(15), Spark(16) and Storm(17). Apache Flink is a new addition to these platforms, as it was announced as a Top-Level Apache project in January, 2015. Flink supports both batch and stream processing.

Graph streaming is a relatively new idea. There has been research done on various applications of graph streaming, as well as on different algorithms that can be done on streamed graphs. However, to our knowledge, no popular data processing engine that supports streaming has had a Graph Streaming API before.

In this thesis we propose a Graph Streaming API for Flink. It is the stream processing counterpart of Gelly, Flink's own batch processing graph API. This API provides methods that facilitate the creation of graph streaming algorithms, providing high

performance and clean code. We surveyed the current state of research and looked at some of the proposed algorithms. Using our API it is be easier and faster to implement these algorithms on Flink, while maintaining all of the advantages that the data processing engine provides, such as fault tolerance, easier development and task management.

## 1.2 Contributions

We have created a new graph streaming programming model, and an API on top of Flink streaming that provides similar functionality to Gelly, where applicable.

This programming model (and API) allows the user to apply basic transformations, such as the mapping and filtering of edges or vertices. Moreover, it provides basic statistics, such as different degree counts (in-, out- and total), total vertex and total edge count.

Other than these basic operators, we propose higher-level features that are useful when developing graph streaming algorithms. Since the need to aggregate results computed by parallel sub-tasks comes up often, we include special *aggregation* functionality, specifically suited for incremental graph processing. This hides the details of manual state merging from the user, so all they have to supply is the core application logic itself.

As an addition to the aggregate method, we have created a *global aggregate* that combines all previously computed results in a single sub-task. This is required in functions that compute global statistics of the streamed graph, such as the total number of edges.

Another feature we propose is an extension of the previously mentioned aggregate method. For algorithms where merging two partial results is significantly easier than merging all $n$ of them, we propose a *merge-tree* abstraction. This operator will arrange a binary tree of mappers after the initial parallel sub-tasks. As shown in the evaluation section, the performance can be facilitated by further stream discretisation, such as windowing, which is already present on Apache Flink.

All of the API methods we provide aim to utilize the distributed nature of the execution environment as well as possible.

Finally, we implemented several distributed graph streaming algorithms using the developed API and measured their performance. We converted the original, centralized streaming algorithms to distributed versions. Our graph streaming API, along with all the examples is open-source, and can be found at `https://github.com/vasia/gelly-streaming`.

## 1.3 Results

One factor we had to consider throughout our work was the distributed nature of Flink. All centralized algorithms we implemented had to be changed in order to support distributed execution. Some of these algorithms could not meaningfully be converted to a distributed version.

The majority of the tests we completed were done in a local environment. However, some of tests were done on a cluster, in order to evaluate the performance of the distributed algorithms on large graphs.

The results of our experiments show that the abstractions we created, such as the distributed merge-tree are indeed a well-performing alternative to the naïve implementation, and they are able to process large graphs in a distributed environment with superior performance.

## 1.4 Structure of the Thesis

Section 2 gives the necessary background of streaming in general, the ways to stream graphs. It also mentions the related graph streaming papers that we based some of our work on. Chapter 3 introduces Apache Flink, its streaming library and Gelly, the batch graph processing library in Flink. Section 4 discusses the different algorithms we implemented as a part of this thesis, to gain insights on what operations a graph streaming API needs to cover.

Section 5 specifies the details of our proposed graph streaming API. In section 6, we discuss some of the important implementation details of our work. The evaluation section 7 describes the ways we tested the algorithms implemented with our API, how the environment was set up, what data set we used and what the results are.

Finally in section 8 we conclude the thesis report by discussing the graph streaming API we created, it's up- and downsides, as well as directions for future work.

# Chapter 2
# 2 Related Work

## 2.1 Streaming as a Programming Model

Stream processing as a paradigm is not a new concept by any means. Streaming in a distributed environment has become a popular and efficient programming model for large scale computations that need to process a large amount of streamed input in real-time.

In stream processing, data arrives as a continuous stream. This data can be as simple as temperature values, or something complex, like tweets that use a specific hashtag. The key constraint behind all streaming algorithms is that the whole stream cannot be stored in main memory, or even on the disk. As such, the streaming model only allows a constant amount of storage, that is not related to the number of input elements.

There are two main models of processing a data stream. The first model maintains a condensed state during the execution of a streaming algorithm (also referred to as *synopsis*). A good example for a synopsis is the state we store when calculating an average of numbers — instead of keeping each element in a list, we can simply store a sum and the number of inputs involved in that sum. Condensing state usually leads to algorithms that provide an estimate instead of accurate results.

The other way to extract information from a stream is through the use of windowing. We gather values into a buffer — the window — and when it is full it will be *triggered*, and we process all of the values. After that, the window is *evicted*, removing some or all of the elements from the buffer. Windows that evict all elements after they are triggered are referred to as *tumbling*, while when some elements are kept we call it *sliding*.

Windows can be defined by their size (number of elements) or by time constraints, where we gather values from a specific time frame (the last $n$ seconds, minutes, hours, etc.). An example of this is an algorithm that computes the highest temperature for every hour.

Another possible method of windowing is the *policy based* approach. Here, the user supplies a custom function that defines when to trigger, and how to evict a window, based on the elements in the stream. For example, if we are working with stock values, we can define a policy that triggers when the price difference between the first and current value exceeds a threshold.

## 2.2 Graph Streaming Research

Before deciding on a graph streaming model and API, we evaluated the current state of the art on graph streaming research. This section describes some of the key results of previous research on graph streams, along with a collection of algorithms which are possible on streams of edges. These algorithms were used to drive our development process, as we used the insights gained during their implementation to form our graph streaming API.

### 2.2.1 Incidence streams

Graph streaming in all related work was defined by a single, finite stream of edges. A special kind of streaming that only applies to graphs is *incidence* streaming (9). In incidence streams, all edges that are connected to the same vertex will be processed at the same time, appearing next to each other in the stream. This also means that every edge appears exactly twice. Incidence streams carry a greater amount of information than regular, randomly ordered streams of edges. They are a rather rare form of graph stream, as not everything can be processed in this fashion. An example of this could be a special way of web-crawling, where we only visit web pages once and consider the graph of the web static.

### 2.2.2 A Semi-streaming Model

The streaming model is not very suitable for graph streaming algorithms as its storage limitations are too strict, permitting only constant size memory. For example, the storage of all vertices is not allowed, since it is normally related to the number of input elements. Using this regular streaming model, we would not be able to count the distinct number of vertices.

One result of previous research is the idea of a semi-streaming model (10) (14). Here we are allowed to store data that is logarithmically related to the number of edges in the stream. This is referred to as *polylog* — $polylog(n) = log^C(n)$, where $C$ is constant. Another definition permits $O(m \cdot polylog(m))$ space, where $m$ is the number of distinct vertices in the data stream.

Throughout this report we define graphs as $G = (V, E)$. This model is only feasible if the graphs that we have to process are similar to real-world networks, where $|V| \ll |E|$ applies (18). It certainly does not work with very sparse graphs, such as trees, since there $|E| = |V| - 1$ stands and we can not store $O(|E|)$ elements.

Finally, in the regular streaming model each element is processed once. Some graph algorithms are not solvable with a single pass, so the semi-streaming model also permits $O(polylog\, n)$ passes.

## 2.3 Graph Streaming Models

What constitutes a graph stream is not a trivial question. There are several ways of representing a graph stream, each bearing pros and cons. We will first enumerate them in this section along with their limitations to motivate our final design.

**Combining edge and vertex streams** The first approach was to have separate edge and vertex streams. When we need to know the vertex values corresponding to the end points of an edge, the two streams can be joined. Since streams are continuous, this can only happen in windows. This model seems problematic, as having to use windows everywhere results in a lot of possible issues and limitations. Moreover, having a separate stream of vertices is not very realistic for a graph stream — in most use-cases the vertices are either known beforehand or streamed as part of the edges.

**Triplet stream** Another way to stream graphs would be the use of triplets. Triplets are pairs of vertices around an edge. Both the edge and the two vertices can have their own value. Having separate vertex values is a great asset for algorithms that utilize them, although these values can be stored in the edge value as a $(Value_{src}, Value_{trg})$ pair. This approach seems less problematic, but there is one issue remaining — what happens when we do not know both vertex values at the time the triplet should be streamed? We could stream incomplete triplets, with some missing vertex values, but this raises a lot of further unanswered questions.

**Edge-only stream** The final model we considered is the simplest. The graph consists of a single stream of edges. Algorithms that use vertex values can still be implemented on this model. And it does not seem to offer severe limitations. This is also the model that was used in most previous research papers on graph streaming.

The main difference between the research mentioned in (14), (20) and (10) and our approach is that we consider truly unbounded streams of data. This is because we want to create a processing model that works on constantly evolving graphs, not on static graphs that simply cannot fit in memory. The latter problem is already covered by Gelly.

**Assumptions** To conclude, our graph streaming model consists of an unbounded, continuous edge-only stream, permitting $O(polylog\, n)$ memory space and $O(polylog\, n)$ processing time for each edge, if $n$ represents the total number of non-distinct edges in the stream. Furthermore, in some algorithms (9), we have to assume that we know all vertex keys in advance.

As a result, multiple-pass algorithms are not possible to implement in our model, since there is no structured loop/iteration present in the streaming programming model of Flink. If structured loops, or iterations over windows were implemented, multiple-pass algorithms would be possible over windows. Multi-pass algorithms are possible to be implemented on static graphs or snapshots of streaming graphs. Moreover, we can not assume that we will have enough memory/disk space to keep the full data stream for one pass stored.

The semi-streaming model allows for the storage of $O(polylog\, n)$ elements for $n$ edges, and also $O(polylog\, n)$ passes. Since we cannot have multiple passes in our model, we use a modified semi-streaming model. Not every algorithm of the semi-streaming model will be possible to implement in our model.

### 2.3.1 Algorithms

Several graph streaming algorithms have been introduced in the literature (14, 10, 20, 9, 8), We select bipartiteness, weighted matchings and triangle counting to explore in detail.

**Bipartiteness** The bipartiteness algorithm decides whether a graph's nodes can be separated into two distinct groups, such as no edge connects any two vertices inside the same group. (10) presents a streaming algorithm to decide bipartition that requires $O(|V|)$ space. It is a great fit for a streaming model, because after the graph has been proven not to be bipartite, the entire computation can terminate.

**Matchings** Single pass, approximating algorithms exist for both weighted and un-weighted streaming graph matchings. An un-weighted greedy matching algorithm provides a 2-approximation (14), meaning that the result will be within the bounds of $[\frac{1}{2} \cdot E, 2 \cdot E]$, where $E$ is the actual value we are approximating. The simplest weighted version produces 6-approximations (10).

**Triangle counts** In (9), Buriol et al. present 1-pass algorithms for both random and incidence streams, to estimate the number of triangles in a graph. The results are very impressive for incidence streams, where we stream all edges connected to a vertex at the same time. This, however, is not the most realistic scenario. Results are rather sub-optimal for normal, randomly ordered streams of edges, with error rates up to 40%, even with a decent number of samples.

Slightly different from the previous algorithms, (8) estimates triangles connected to each node in a large streamed graph. This is a much more useful metric, since it can

be used to estimate other statistics, such as local (and as a result global) clustering coefficients.

# Chapter 3
# 3 Apache Flink

Apache Flink is a distributed general-purpose data processing engine that is fast, reliable and scales well. It exploits in-memory processing whenever possible in order to offer very high processing speeds, but it is also designed to perform well when the available memory is not sufficient for in-memory execution.

## 3.1 Flink Overview

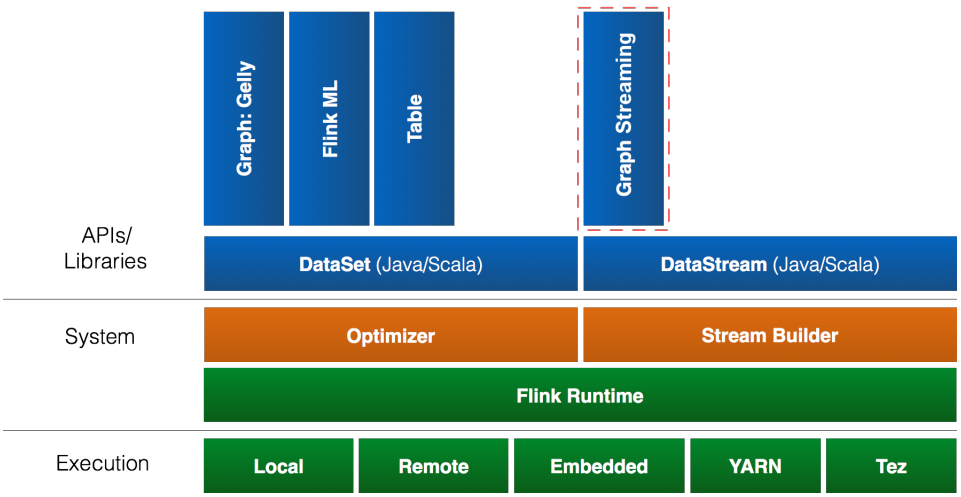



*Figure 3.1: The Flink Stack*

An overview of the Flink stack is shown on figure 3.1. The dashed lines show where our graph streaming API could be placed in this stack.

There are multiple ways to execute Flink programs. Local execution can be used for debugging and testing, but when performance is important, programs should be executed on YARN or a standalone cluster.

The Flink runtime handles job- and task managers. Job managers take care of scheduling and resource management. They deploy tasks on the task managers and receive their status updates. Task managers handle the actual task execution, and exchange intermediate results among each other whenever it is needed.

## 3.2 Flink Streaming

Flink has its own Stream Processing API, offering high-level functions operating on data streams. There are many different data sources available, such as file sources, web sockets and message queues. Moreover, users can define their own data sources with ease. The most common type of data stream consists of *Tuples*. Flink has a custom implementation and wide-spread support for tuple types.

When we want to manipulate a data stream with Flink Streaming we simply have to apply transformations to it. There are many different types of transformations, such as the `map`, `filter`, `fold`, `reduce` operators.

**Grouping** Streams can also be grouped using the `groupBy` operator, resulting in a different type of stream. A grouped data stream supports aggregations by key. The `groupBy` function takes a *key selector* as its parameter. This defines how the stream should be broken up into separate groups. Key selectors can be purely positional, or they can group data based on its value. As an example, if a stream consists of tuples with 2 elements and we group by the first elements, we will get a group for each distinct value encountered.

**Partitioning** Partitioning is a very important part of distributed streaming algorithms. The correct method of partitioning can greatly affect the performance of a solution. There are a set of built-in partitioning methods supported by Flink. It is possible to create a custom scheme by changing the visibility of a few internal Flink methods.

**Discretisation** In Flink, data streams can be discretised into windows. There are various ways to define an eviction policy, which describes how windows should be emptied and processed over time. This can be based on the number of elements in the window, based on time, or based on user-defined metrics.

**Iteration** Flink streaming also supports *iterations*. To use iterations, we have to define a *step function* that works on the iterated stream. Then we can define the *head* and *tail* of the iteration, and close it with any stream of data. This stream that we use to close the iteration will be fed back to the iteration head in the next round.

## 3.3 Gelly

Gelly is the graph API in Flink. It works on top of the batch processing API and offers many different ways to analyze/process large graphs. Gelly is based on the Bulk Synchronous Parallel(19) (BSP) model, which is based on structuring loops. In this model, every multi-pass algorithm is possible.

Since we are working on top of Flink Streaming, we cannot directly use Gelly on data streams. Instead, we opted to create our own streaming graph abstraction, that will be similar to graphs in Gelly.

**Basic operations** Gelly supports many different operations on graphs that can be used for streamed graphs as well. Mapping the values of vertices or edges, counting the degrees of vertices, counting the numbers of vertices or edges, and getting the union of two graphs are examples of such functionality.

**Iteration** There are methods in the Gelly API that cannot be used in a streaming environment. Any method that uses iterations, such as the vertex-centric and gather-sum-apply iterations, will be impossible, since we have to deal with unbounded data streams. Another key part of Gelly we can not use is joins. As a result, we will not have an equivalent to the *joinWithVertices* and *joinWithEdges* functions.

**Neighborhood methods** Neighborhood methods are possible for vertices, albeit only with a subset of the real neighbors of each vertex, that we have seen in the stream up to the point of execution. According to our model, edge neighborhood methods should not be possible, because we have to limit our storage to $polylog(n)$ and storing a single neighbor for each edge would already exceed this limit.

# Chapter 4
# 4 Graph Streaming Algorithms

This section introduces 4 different graph algorithms which are feasible and fitting in a streaming environment. These algorithms are degree counting, bipartition, triangle count estimation and weighted matching. We explore the original centralized streaming solutions, explain why they are a good fit for our streaming model, then describe how they can be executed in a distributed environment. The implementation of these algorithms is discussed in section 6.

## 4.1 Graph Degrees

Counting the degrees of vertices is a core functionality of any Graph API. As such, we provide a streaming equivalent of this operation, described in this section.

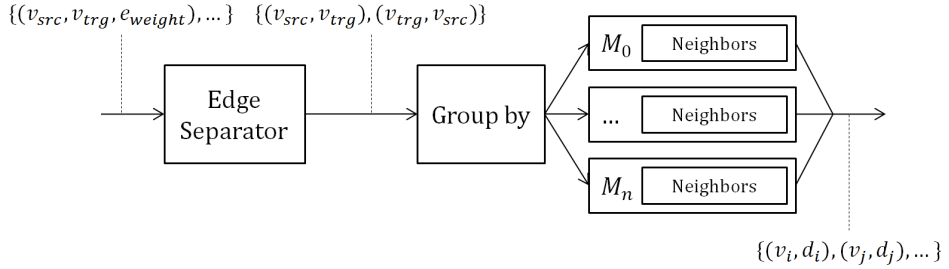



*Figure 4.1: Graph Degrees Algorithm*

Degrees of vertices are important metrics, used in different graph algorithms. In the case of a directed graph, we can differentiate between in-, out- and total degree counts. We implemented all three versions, as they only require minor changes.

If we assume that $|E| \gg |V|$ in our graphs, which is a realistic assumption in real-world networks, it is feasible to store the neighborhood of each vertex. It is not

*Algorithm 1: Streaming Degree Count Algorithm*

```
1: procedure DEGREE COUNT
2:     neighbors = ∅;
3:     degrees = ∅;
4:     upon event new edge do
5:         src, trg ← edge;
6:         if src not in neighbors[trg] then
7:             neighbors[trg] ← neighbors[trg] + src;
8:             degrees[src] = |neighbors[trg|;
9:         if trg not in neighbors[src] then
10:            neighbors[src] ← neighbors[src] + trg;
11:            degrees[trg] = |neighbors[src|;
```

enough to simply keep a counter for each vertex, because our model should support duplicate entries.

This degree counting algorithm is shown in algorithm 1. First we split all edges in up to two records containing the source and/or target vertices, depending on which degrees we want to count. Then we can utilize grouped streams and launch a separate mapper for each vertex after a `group by`, where we collect the current degree count. This process is shown in fugre 4.1. To measure only in-degrees, we simply omit $(v_{src}, v_{trg})$ from the edge separator's output, while to measure out-degrees it is $(v_{trg}, v_{src})$ that we need to remove.

## 4.2 Bipartition

The bipartition algorithm checks whether the vertices of a graph can be divided into two partitions such that there are no edges inside either partition. In other words, the graph is 2-colorable. This also means that there is no odd length cycle in the graph.

As an example, the graph in figure 4.2 is bipartite and the two bounding boxes show two possible partitions. Node 5 could be in either partition and nodes 4 and 8 are reversible.

The bipartiteness check algorithm is a great fit for the streaming model, because it aims to prove that the graph is not bipartite. When the algorithm finds an inconsistency, it can terminate the whole execution, ignoring any further edges in the stream. This would not be possible in a batch model, as we are processing all edges at the same time.

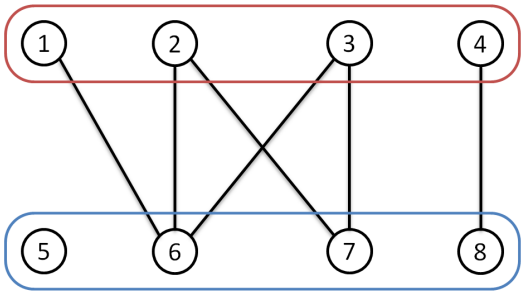
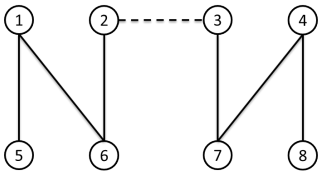



*Figure 4.2: Bipartite Graph Example*




| Before | | |
|---|---|---|
| Vi | Si | Ci |
| 1 | + | 1 |
| 2 | + | 1 |
| 5 | - | 1 |
| 6 | - | 1 |
| 3 | + | 3 |
| 4 | + | 3 |
| 7 | - | 3 |
| 8 | - | 3 |

| After | | |
|---|---|---|
| Vi | Si | Ci |
| 1 | + | 1 |
| 2 | + | 1 |
| 5 | - | 1 |
| 6 | - | 1 |
| 3 | - | 1 |
| 4 | - | 1 |
| 7 | + | 1 |
| 8 | + | 1 |

*Figure 4.3: Streamed Bipartition Candidate Example*

### 4.2.1 Centralized Solutions

Implementing a centralized version of the bipartition algorithm on a non-streamed graph is straightforward - we need to traverse the graph using the Breadth-First Search algorithm, and assign alternating colors, based on the level of the tree we are on. When we see a conflict, the graph is proven to be non-bipartite.

Evaluating a streamed graph is not so simple - we don't have random access to edges. Feigenbaum, et al. (10) describe a 1-pass semi-streaming algorithm that solves bipartition on streamed graphs.

The solution is to maintain the connected components of the streamed graph at every step. Vertices within each component are assigned a sign, such that neighboring vertices can never be assigned the same sign. When an edge joining two previously disjoint components arrives, we can join these components. To resolve the arising conflicts, signs may have to be flipped in one of the components. If we find an inconsistency during this process, the graph is proven not to be bipartite and the algorithm terminates.

Table 4.3 demonstrates how components can be merged in the streamed bipartition algorithm. In the example, the current edge being processed is (2, 3). Before the

edge arrives, we store information on two components in the table. When we receive the edge we are able to merge these two components. As both 2 and 3 have the sign +, we can not merge the components right away - the signs in one of them have to be flipped. When there are more edges to merge by, we can run into inconsistencies, which show us that the graph is not bipartite.

The pseudo-code for this algorithm is shown in algorithm 2. The `combine` function is further detailed in 6.1.1.2.

*Algorithm 2: Streaming Bipartition Algorithm*

```
1: allComponents = ∅;
2: procedure BIPARTITION
3:     upon event new edge do
4:         touchedComponents = ∅;
5:         for all component ∈ allComponents do
6:             if edge connected to component then
7:                 touchedComponents ← touchedComponents + component;
8:         allComponents ← combine(touchedComponents);
```

### 4.2.2 Distributed Solution

When the bipartition of a large graph stream needs to be evaluated, we may want to consider running the algorithm in parallel. This means that several instances would each process a separate chunk of the edges. Later, the output of the parallel instances has to be merged. If the graph is dense, the output of the instances will be significantly smaller than their input, since we only store one value per vertex. As a result, in a dense graph, nodes responsible for merging state will receive much less data than the total input. The difference between input and output sizes depends on the ratio between $|E|$ and $|V|$, assuming degree distribution in the graph is not dramatically skewed.

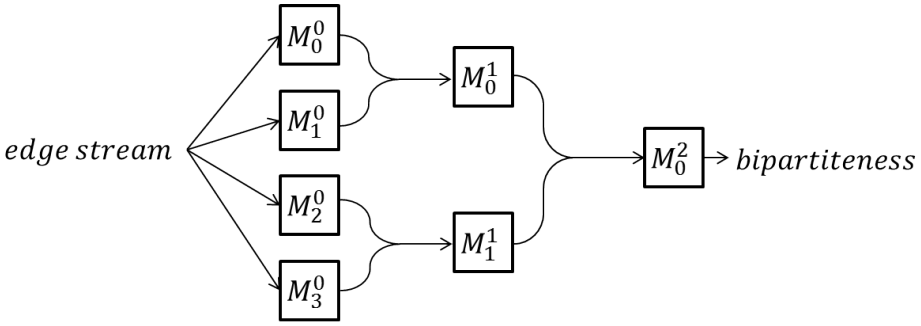



*Figure 4.4: Distributed merge tree structure*

To merge the output of the parallel instances, we use a *merge-tree* structure, shown in figure 4.4. When each mapper receives the output of exactly two other mappers, for any degree of parallelism $p$ the tree will contain $2 * p - 1$ nodes. This ratio between mappers could be increased to avoid communication overhead.

The data structure (*candidate*) that is processed by the mappers is complemented by the identifier of the mapper that produced the current output. This needs to be propagated through the tree, in order to achieve the tree structure by grouping by this identifier.

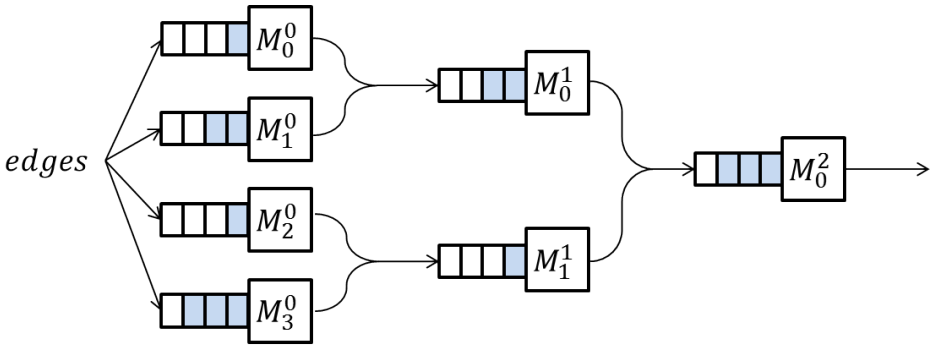



*Figure 4.5: An improved merge tree to reduce communication overhead*

With the arrival of each new edge, we need to update the state of all mappers in a path to the top of the merge tree, affecting $log(p)$ nodes. This causes a lot of communication overhead. One possible solution is to use windowing. Figure 4.5 shows how this could be achieved. This way, mappers would only trigger once their window is full (or the stream has ended). Between any two levels of the tree, if the size of the window is $w$, we avoid $(w - 1) \cdot p$ unnecessary updates. The size of the windows should decrease towards the root of the merge tree.

## 4.3 Estimated Triangle Count

An estimate for the number of triangles in a graph is a useful metric that can be used to reason about the global structure of the graph. For example, the global clustering coefficient or the transitivity ratio can be calculated using the global triangle count.

This problem has been very well studied in batch and/or centralized environments and there are a couple of papers that deal with streamed, distributed triangle counts. In this section, we describe two different ways to estimate the number of triangles in large streamed graphs. One of these algorithms estimates the total number of triangles, while the other solution counts triangles for each vertex in the graph.

$$\widetilde{T_3} := (\frac{1}{s}\sum_{i=1}^{s}\beta_i) \cdot |E| \cdot (|V| - 2)$$

*Figure 4.6: Relationship between the sampling results and estimated triangle count*

$$s \geq \frac{3}{\epsilon^2} \cdot \frac{|T_1| + 2 \cdot |T_2| + 3 \cdot |T_3|}{|T_3|} \cdot ln(\frac{2}{\delta})$$

*Figure 4.7: Instance count and approximation quality*

### 4.3.1 Global Triangle Count Estimate

In (9), the authors describe a 1-pass, space bounded algorithm that is able to give an estimate for the total number of triangles in a graph, and also scales well. The results are very accurate for incidence streams (where edges incident to a vertex are streamed at the same time), and somewhat accurate, but in some cases sub-optimal for regular (arbitrarily ordered) edge streams.

The estimate is gathered using a random sampling method called *reservoir sampling*. This solution gives us uniformly random elements over an unbounded stream. With every update, the newly arrived edge is sampled with a probability that decreases over time. The probability of sampling an edge when $i$ edges have already been processed is $P = \frac{1}{i}$.

Along with the edge, we sample a random vertex in the graph. This means that we need to know the vertex identifiers in advance. When we choose not to sample an edge, we check whether it connects the sampled edge to the previously selected vertex. If we find both such edges, we have found a triangle and the result of the sampling is 1, otherwise it's 0. This result is referred to as $\beta$. Since we are working on an edge stream, we may decide to sample a new edge over time, in which case $\beta$ will again be 0, until we find the edges connecting it to the currently sampled third vertex.

Many instances of this sampling algorithm have to be executed in order to get accurate results. The more instances we run, the more accurate the results will be. The relationship between the sampling results and the estimated number of triangles in the graph is shown in the equation in figure 4.6. In this equation, and for the rest of this document $s$ indicates the number of sampling instances. Figure 4.7 indicates how good an estimate can be with a given number of instances. If $s$ satisfies the equation, we will get an $(1 + \epsilon)$-approximation with probability $(1 - \delta)$.

We implemented this algorithm on Flink streaming with some necessary changes. Since the stream might be infinitely long, we wanted the results to be emitted

continuously so they get more and more precise over time.

Making the algorithm distributed is very important for performance. The naïve way to distribute the load is to run the sampling algorithm on many parallel sub-tasks and simply send all the edges to all sampling instances. This causes some communication overhead, but the time and space required to run the algorithm will be the same per instance, resulting in faster overall execution.

Another possibility is to create a custom partitioner which keeps track of the currently sampled edge for each instance, and decides where to route the edge based on this data. This means that some edges will be sent to multiple sub-tasks, although this will be rare after a large number of edges have been processed. Using this method, we can be sure that no information is lost that would otherwise be visible in a centralized solution, and at the same time we avoid sending some of the edges to the sampling part of the execution.

### 4.3.2 Local Triangle Count Estimate

In (8) the authors describe a way to estimate triangles incident to each vertex. This metric is even more useful than the global triangle count estimate, because we can use it to calculate an estimated clustering coefficient, and any other metric associated with local triangle counts.

They propose an algorithm that requires the storage of only one counter per node in main memory. Unfortunately, the solution requires multiple passes over the input stream, which is not directly possible in our model.

## 4.4 Weighted Matchings

A matching in a graph is a subset of the edges, inside which no two edges are connected. All graph matchings produce a bipartite graph, but this is a much stronger constraint than bipartition.

There are two types of graph matchings — un-weighted, and weighted, depending on whether the graph's edges have values or not. The goal when creating an un-weighted matching is to include most of the edges. Weighted matchings are more complicated. Here, we want the total sum of values of edges present in the matching to be maximal. Un-weighted matchings are a special case of weighted matchings, where all edge values are 1 (or any constant number greater than 0).

### 4.4.1 Centralized Weighted Matching

A 1-pass, centralized weighted matching algorithm is introduced in (10), which results in a matching that has at least $\frac{1}{6}$ of the optimal weight. The algorithm is very simple. For each edge with weight $w(e)$ we calculate the sum of weights from the *colliding* edges already in the matching, $w(C)$. A colliding edge is one that shares either endpoint with another, since they cannot be present in the same matching. If $w(e) > 2 * w(C)$, we replace the colliding edges with the new one, otherwise we ignore the new edge. The space complexity of this algorithm is $O(V)$, since we have to store at most one edge for each pair of vertices. With the correct data structures the time complexity of the algorithm is $O(1)$, since we only have to look up 2 edges in the current matching in each iteration.

### 4.4.2 Distributed Weighted Matching

While it initially seemed straightforward to convert the centralized algorithm to a distributed one, it turned out to be a difficult problem, that may not be ideal in our model. Figure 4.8 summarizes the structure that could possibly solve weighted matching in a fully distributed manner, although this would include a lot of overhead.

The main idea comes from the fact that each edge may have at most 2 colliding edges. We can send each edge to two separate sub-tasks, a master and a slave. The partitioner would make sure that all edges with the same source vertex are routed to the same master, while all edges with the same target are directed to the same slave. With such partitioning, we could collect all colliding edges with one step. The collisions we find would be sent to a single sub-task in the next mapper. This is where the decision on keeping the edge can be made. If the edge is kept, collisions have to be discarded.

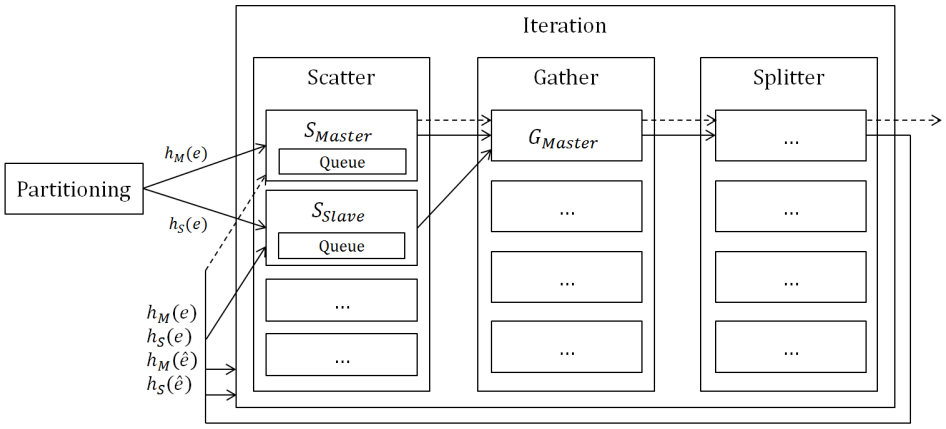



*Figure 4.8: Distributed Weighted Matching Scheme*

A maximum of 6 messages have to be sent to the first mapper — to do this, we have to utilize iterations. We need 6 messages at most, because we have to reach the master and slave of the new edge (2), as well as the master and slave of the colliding edges ($2 * 2$). Upon receiving the messages, the sub-tasks in the first mapper change their local matchings, removing any instances of the collisions and adding the new edge in the original master and slave.

Only after this can we finalize the addition of the new edge and removal of the collisions, and send them outside of the iteration, where they will be processed, and a new total weight will be calculated. This is represented by the dashed lines in the figure.

**Limitations** However, this solution still suffers from a problem. Edges arrive in a continuous stream, but we have to wait for the possible addition of each edge to be finalized before we can start processing the next edge. If we omit this, the local matchings will be inconsistent. This constraint means that the first mapper has to be "locked" while there is an edge passing through the iteration. As a result, we have to maintain a queue of newly arrived edges. This goes against the streaming philosophy, and most probably causes too much overhead for the algorithm to be plausible.

Nevertheless, it was useful to consider the distributed implementation of this algorithm. It is apparent that even the simplest centralized algorithms may be very difficult to distribute in a streaming environment. Having to use an iteration to "communicate" between mappers seems to be a sign that the algorithm may not be the best fit for the streaming model.

It should be noted that this problem can also be solved by merging the partial results of parallel sub-tasks with a merge-tree. This approach would not be as interesting in our case, since we already have an example usage of the merge-tree, and this algorithm stores even less complex state than bipartition.

Finally, a completely different approach could also be possible, that offers a different quality of approximation (as opposed of the 6-approximation algorithm described in the papers). However, we decided not to explore this path any further.

# Chapter 5
# 5 Graph Streaming Framework

## 5.1 Discussion of Implemented Algorithms

We implemented several graph streaming algorithms in order to better assess what is needed for a graph streaming API. In this section we draw the conclusions from the design and implementation of these algorithms.

### 5.1.1 Bipartition

The merge-tree that we used to implement a distributed version of the bipartition algorithm could be useful for many other distributed streaming algorithms where different instances of state need to be merged. With more complex state, a merge-tree solution can be even more beneficial, since we always only merge 2 items into 1, instead of $n$ to 1.

Using this structure with windowing is highly recommended, because the windows reduce a great amount of network traffic and unnecessary merging of state. This comes with the cost of having a reduced output granularity — while without windowing we can return a partial result after the arrival of every edge, using windows decreases this rate. There may be algorithms where this output rate is critical. Any procedure that has to produce real-time results may be an example of this. Without the use of windows however, the merge-tree is not a scalable construct.

### 5.1.2 Global Triangle Counts

While this algorithm may not be the best fit for Flink's streaming architecture, it does offer some insights. The idea of using sampling to convert a multiple pass algorithm to a one pass version is very interesting, and having a method to sample the stream within the API would be very useful for any algorithm that calculates estimates.

Our first distributed implementation for the algorithm is based on the very simple idea of broadcasting every edge to every parallel sub-task. As a result, the sub-tasks only have to run their share of the sampling instances. This solution will be referred to as *broadcast* in the rest of the document.

As proposed in 4.3.1 another possible way to make this algorithm distributed would be by using a custom partitioner, where the sampling happens. This implementation is described in more detail in chapter 6.

This custom method of partitioning could be part of our API, because it can be useful for any algorithm that samples edges. The main idea is to move the reservoir sampling to a partitioner, and keep track of multiple sampling instances there. When an edge arrives, we can flip a coin for each instance just like in regular reservoir sampling. If we decide to re-sample an edge, we emit it to a particular sampling instance. When the coin flip fails, we only emit the edge to the sampling instances where the currently sampled edge is connected to the newly arrived one. Since the number of samples that we run is constant, most of the edges can be discarded. Moreover, because we use reservoir sampling, where the probability of re-sampling an edge gets lower over time, this rate of edge discarding increases as the stream goes on.

If this was a part of our framework, we could hide the details of handling separate sampling instances on separate sub-tasks from the users. As a result, they would only be required to write the core of the sampling logic. Two possible flavors of this sampling technique could be created — one where we only send the sampled edge itself, and one where we send all edges *incident to* the sampled edge. By incident edges we refer to two edges that share a common endpoint. In the rest of this document, where we describe our second implementation of the triangle estimation algorithm, we will refer to this as *incidence-sampling*.

The more complicated the algorithm that does the sampling is, the more benefits incidence-sampling will provide, since the sampled edges can be processed in a fully distributed environment.

## 5.2 Graph Streaming API

This section describes the core functions of our graph streaming API, along with their usefulness and potential drawbacks. We also discuss some of the planned, unimplemented methods.

In its current state the API consists of a single class, `EdgeOnlyStream`. There is a single constructor, taking a stream of edges and the context of the execution environment. The reason to support only edge streams has been explained in 2.3. Table 5.1 shows the different methods supported by the API. The following paragraphs briefly explain each method.

| Name | Description |
|---|---|
| `getContext` | Returns the current execution environment |
| `getVertices` | Returns a stream of distinct vertices |
| `getEdges` | Returns the stream of edges |
| `mapEdges` | Applies a mapper function to all edges |
| `filterVertices` | Applies a filter function to all the vertices of arriving edges |
| `filterEdges` | Applies a filter function to all arriving edges |
| `distinct` | Returns a streamed graph with distinct edges |
| `reverse` | Returns a streamed graph with reversed edges |
| `union` | Returns the union of two streamed graphs |
| `undirected` | Returns a streamed graph where edges are present in both directions |
| `numberOfVertices` | Returns a stream with the number of distinct vertices |
| `numberOfEdges` | Returns a stream with the number of streamed edges |
| `getDegrees` | Returns a stream of the total degrees of all vertices |
| `getInDegrees` | Returns a stream of the in-degrees of all vertices |
| `getOutDegrees` | Returns a stream of the out-degrees of all vertices |
| `aggregate` | Aggregates the output of a mapper for each edge or vertex |
| `globalAggregate` | Creates a global aggregate, with a parallelism of 1 |
| `mergeTree` | Merges parallel results with a merge-tree structure |

*Table 5.1: Graph Streaming API Methods*

**getContext** This is a simple getter method for the current execution environment.

**getVertices** As explained in 4.1 we can assume that we are able to store the neighbors for each vertex. This is how this method is able to provide a distinct stream of vertices.

**getEdges** This is a simple getter method for the edge stream.

**mapEdges** This method applies a map function to all new edges in the stream and returns a new streamed graph. It is possible to change the type of the edge values here.

**filterVertices** An user defined filter function is applied to both vertices of every new edge in the stream. If both of the vertices pass the filter, then the edge is kept. The result of this method is a new streamed graph.

**filterEdges** A simple filter is applied to each arriving edge in the stream.

**distinct** This method is feasible in our model because we only keep the target vertices for the sources of each edge. If a the target of an edge has been seen before, we do not keep that edge in the resulting graph stream.

**reverse** This reverses the direction of all edge with a simple mapper.

**union** To get the union of two graph streams of the same type, we simple merge their edge streams.

**undirected** The `undirected` method works by unioning a graph stream with its own reversed version.

**numberOfVertices** This method utilizes the `globalAggregate` function and returns a continuously improving stream with the number of distinct vertices seen in the graph so far. We can store $O(|V|)$ items in memory, so this method is possible in our model.

**numberOfEdges** It is not possible to store all distinct edges in the graph stream, so this method returns the total number of edges instead.

**getDegrees** All the degree functions use `aggregate` to compute the degrees. A mapper function first splits the edge stream into a vertex stream, then this vertex stream is grouped, and degrees are measured as the sizes of neighborhood sets.

**getInDegrees** Does the same as `getDegrees` but only keeps the record for $v_{trg}$ from $(v_{src}, v_{trg})$.

**getOutDegrees** Does the same as `getDegrees` but only keeps the record for $v_{src}$ from $(v_{src}, v_{trg})$.

**aggregate** Applies a mapper function to all edges in the stream, then creates grouped data streams and aggregates them using another mapper.

**globalAggregate** In some algorithms it is unavoidable to aggregate all results in a single mapper. This is what `globalAggregate` does. Moreover, using a user-specified `collectUpdates` flag, we can make the method only return values that are different from the previously emitted result.

**mergeTree** Builds a merge-tree structure, which is explained in 4.2. The user has to specify the mapper that converts an edge to the aggregate type of the state that is used within the tree, and another mapper that is capable of merging two instances of this state. A third parameter is the size of the window to use when creating the merge-tree.

# Chapter 6
# 6 Implementation

## 6.1 Graph Streaming Algorithms

In this section we give detailed information on the implementation of the graph streaming algorithms we used.

### 6.1.1 Bipartition

#### 6.1.1.1 Centralized Solution

The original algorithm, as described in (10) works in a centralized environment. To implement this algorithm we created a custom type that represents state. This stores separate, disjoint groups of vertices in a map. An example of two such groups is shown in figure 4.3. The dashed edge, $(2, 3)$, connects these groups and after it is processed they will be merged. This example will be used throughout this section.

When an edge arrives, we check whether it connects any two groups, by having its endpoints inside them. If this is the case, we can then attempt to merge these. If we find a conflict during merging, we know immediately that the graph is not bipartite. In our example, the $(2, 3)$ edge connects groups 1 and 3. Each group is identified by the key of its lowest vertex.

When we merge two groups of vertices, we first have to find *all* vertices that lie in the intersection of the groups. The signs of these nodes are the constraints that we *merge by*. These particular signs are either all the same or all different (among the two groups). If this is not the case, we have found an error and the graph is proven to be non-bipartite. Once we know whether we have to reverse signs or not, we can actually complete the merging by adding all new vertices with the correct sign into the local group.

Following the example, the vertices we merge by are 2 and 3. The *Before* table shows us the signs, and we can see that both of them are +. Since we cannot have vertices of the same sign connected in a bipartite graph, we have to change this. The signs in one of the groups have to be inverted.

#### 6.1.1.2 Distributed Solution

The centralized version is easier to implement than the distributed one, because only one edge arrives at a time, and the large state of parallel sub-tasks does not have to be merged.

In the distributed solution, we have to *combine* these maps of vertex groups. Let's follow this process through an example, with the groups shown in figure 6.1. The 2 groups on the left constitute the local map that is stored in memory, while the other side represents the newly arriving map. We can see that in this case, each edge in the incoming map may connect multiple groups in the local one.

In order to merge these maps, we have to compare all of their groups with each other and look for vertices in their intersections. Any two groups with a non-null intersection need to be merged. For example, $Group_0$ of $Map_1$ will be merged into $Group_0$ of $Map_0$, because vertex 2 is in their intersection.

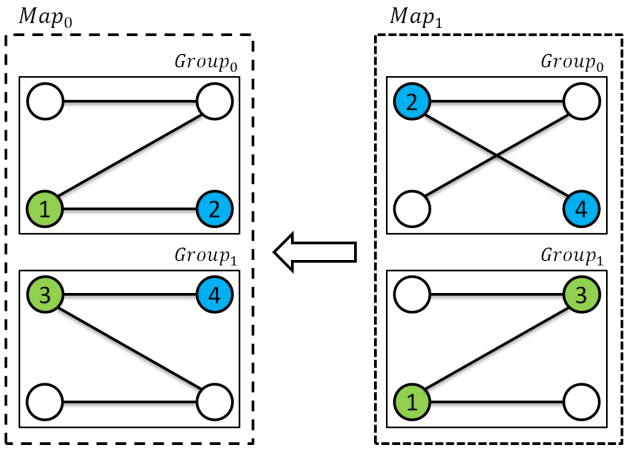



*Figure 6.1: Broadcast Group Combination*

In figure 4.5 we have already shown the high-level model for the distributed version of this algorithm. Using the merge-tree function in our API we simply have to create two functions. The first one maps edges to Candidate objects, and the second one combines candidates with each other. The implementation of the merge-tree itself is described in 6.2.

### 6.1.2 Global Triangle Count Estimation

#### 6.1.2.1 Broadcast Solution

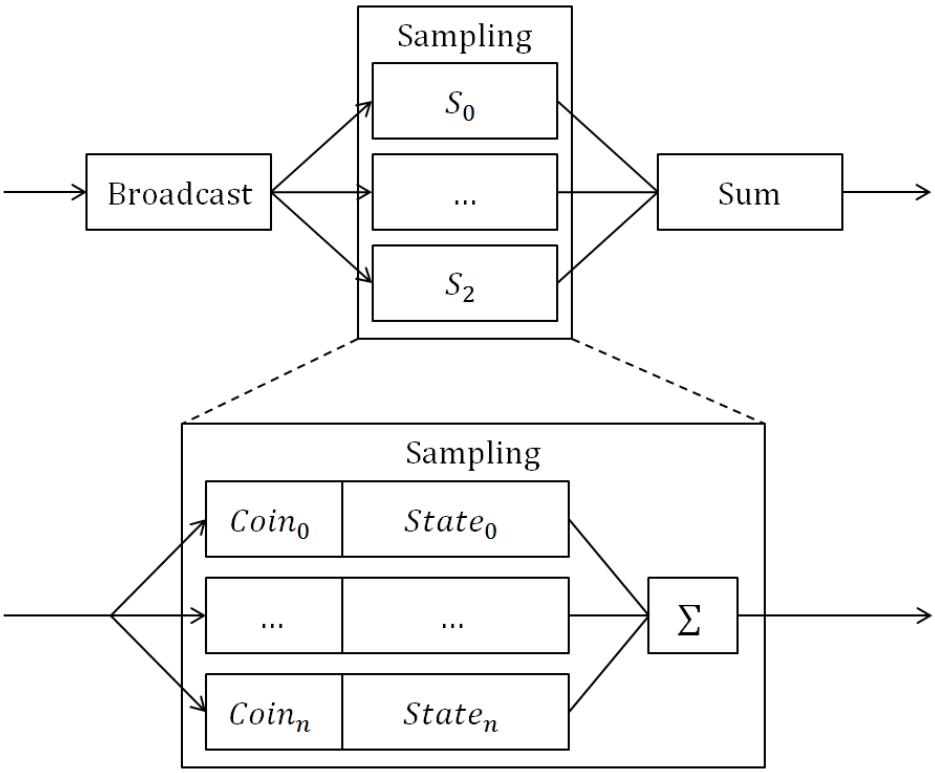



*Figure 6.2: Broadcast Triangle Count Estimation Model*

As pictured in figure 6.2 the model of the broadcast triangle estimation algorithm is rather simple. We send every edge to every parallel sub-task. Inside the sampling mapper we keep a list of coins and sampling states that we work on. After the arrival of every edge, we need to iterate this entire list, flip every coin and update the states where the coin flip was successful. During the iteration we calculate a local sum of $\beta$ values, which we return as the output of the mapper. These values are then summed by another mapper, which has a parallelism of 1.

This solution is not particularly fast. This might be caused by the overhead of having to iterate a very large collection (size $s/p$) of triangle sampling state every time an edge arrives in every sub-task.

To implement this over Flink, we can use the `broadcast` function on the edge stream. This makes sure that all parallel sub-tasks in the subsequent mappers receive every edge. As a result, we only have to process $s/p$ instances in each mapper. After this, a flat mapper is applied to this broadcasted data stream. This is the main

sampling mapper. It maps an edge into a custom class, `TriangleEstimate`. This contains information on the number of edges seen so far in the stream and the new $\beta$ result. The edge and vertex count is required for the calculation of the total triangle estimate. We assume that we know the number of vertices before execution.

In the next stage, a mapper with a parallelism of 1 sums the values that arrive from all parallel mappers. It stores a map structure in order to remember which result came from which mapper. The values in this map are summed after each update, and if the new triangle count estimate is different from the previous result, we emit it.

#### 6.1.2.2 Incidence-Sampling Solution

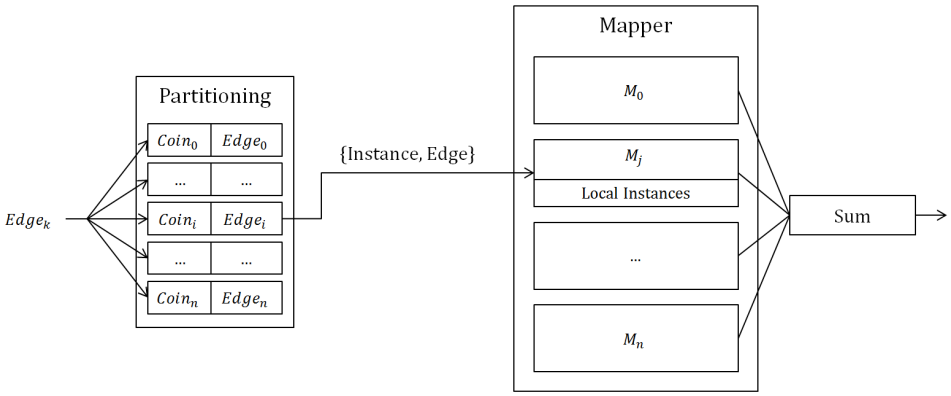



*Figure 6.3: Incidence-Sampling Triangle Count Estimation Model*

A more sophisticated way to run a large amount of sampling instances is through the use of the previously described incidence-sampling technique. Here, a partitioner keeps track of all the sampled edges across all running sampling instances. When an edge arrives, it flips every coin and emits a new event for each successful coin flip. Furthermore, when an edge is connected to any other currently sampled edge, it is forwarded to that sampling instance. The partitioner keeps track of the sub-task that handles each instance. As a result, we can *address* the instances with a particular edge, and avoid having to iterate large list in every step.

## 6.2 Graph Streaming API

This section explains some of the implementation details of our graph streaming API. It will cover some of the more complex algorithms, such as the `aggregate`, `globalAggregate` and `mergeTree` functions.

**aggregate** This is a public method in the API, but it is also used internally by the degree functions. It works by applying a flat map function on all the edges and converting them to any number of vertices with a custom type. These vertices are then grouped by their key value and a map function is applied on all members of each group.

This may not be the most flexible solution, as we may want to apply an aggregate on edge values. If so, we need to have `Edge` as the vertex type, which causes unnecessary overhead. A separate `aggregate` function that works on edges may be added to the API in the future.

Listing 6.1 shows the implementation of the aggregate method. It takes two mappers as its arguments. The first one converts edges into vertices, and the second one maps the vertices that belong to the same group, after the group by operation.

*Listing 6.1: Aggregate implementation*

```
// GraphStream.java
public <VV> DataStream<Vertex<K, VV>> aggregate(
        FlatMapFunction<Edge<K, EV>, Vertex<K, VV>> edgeMapper,
        MapFunction<Vertex<K, VV>, Vertex<K, VV>> vertexMapper) {

    return this.edges.flatMap(edgeMapper)
            .groupBy(0)
            .map(vertexMapper);
}
```

**globalAggregate** The global aggregate function is similar to `aggregate`, but it works with a parallelism of 1, so all results are merged in a single sub-task. When the user specifies the `collectUpdates` flag as true, only distinct outputs are emited. This is done by storing the last result in memory and comparing the current output to it before collecting a value.

The implementation of this function is shown in listing 6.2. The function takes 3 arguments — a function mapping edges to zero or more vertices, a mapper between all of the vertices, and the flag described above.

*Listing 6.2: Global aggregate implementation*

```
// GraphStream.java
public <VV> DataStream<VV> globalAggregate(
        FlatMapFunction<Edge<K, EV>, Vertex<K, VV>> edgeMapper,
        FlatMapFunction<Vertex<K, VV>, VV> vertexMapper, boolean
            collectUpdates) {

    DataStream<VV> result = this.edges.flatMap(edgeMapper)
            .setParallelism(1)
            .flatMap(vertexMapper)
```

```
            .setParallelism(1);

    if (collectUpdates) {
        result = result.flatMap(new GlobalAggregateMapper<VV>())
                .setParallelism(1);
    }

    return result;
}
```

**mergeTree** To organize parallel sub-tasks into a tree structure, we first run every edge through a wrapping mapper. This applies the edge-to-state mapper by the user to the edge, and appends the index of the current sub-task to this value, creating a tuple with two elements. his information can be acquired when using a `RichMapFunction` using the `getRuntimeContext` method.

After that, we start an iteration that is executed $\lceil log(p) \rceil$ times. During this phase we create the windows and apply the user-supplied merging function to pairs of mappers. To create a pairing of the mappers of parallel sub-tasks we use a custom `KeySelector`. It selects a set of bits from the original sub-task indices of all inputs. The starting position of this bit mask is shifted to the left as we progress through the merge-tree.

Table 6.1 shows how this works in practice. In this case we are on the first level of the tree, and the bit-mask is ...1110. The selected bits are shown in bold style. The formula that returns the correct result for each level is $id >> (level + 1)$.

| Source sub-task | Source sub-task (binary) | Target sub-task |
|---|---|---|
| 0 | **00**0 | 0 |
| 1 | **00**1 | 0 |
| 2 | **01**0 | 1 |
| 3 | **01**1 | 1 |
| 4 | **10**0 | 2 |
| 5 | **10**1 | 2 |
| 6 | **11**0 | 3 |
| 7 | **11**1 | 3 |

*Table 6.1: Merge-tree Key Selector Example*

The complete merge-tree implementation is shown in listing 6.3, without the custom *MapFunction* and *KeySelector* operators that we implemented. First, the edges are flat-mapped with `MergeTreeWrapperMapper`, which converts each edge (or multiple edges, depending on the user-defined *initMapper* function) to the type that is used within the tree. Along with the user-defined data, it adds the index of the sub-task that processes the element to the output.

After this, we create windows and apply the user-defined *treeMapper* whenever the window is triggered. This happens multiple times, depending on how large the binary-tree is, defined by the current degree of parallelism. After each level, the output of mappers are paired up using `MergeTreeKeySelector`. This key selector uses the previously added sub-task indices to decide which instances to pair up.

*Listing 6.3: Merge-tree implementation*

```
// GraphStream.java
public <T> DataStream<T> mergeTree(
        FlatMapFunction<Edge<K, EV>, T> initMapper,
        MapFunction<T, T> treeMapper, int windowSize) {

    int dop = context.getParallelism();
    int levels = (int) (Math.log(dop)/Math.log(2));

    DataStream<Tuple2<Integer, T>> chainedStream =
        this.edges.flatMap(new MergeTreeWrapperMapper<>(initMapper));

    for (int i = 0; i < levels; ++i) {
        chainedStream = chainedStream
            .window(Count.of(windowSize/(int)Math.pow(10, i)))
            .mapWindow(new MergeTreeWindowMapper<>(treeMapper))
            .flatten();

            if (i < levels - 1) {
                chainedStream = chainedStream.groupBy(
                    new MergeTreeKeySelector<T>(i));
        }
    }

    return chainedStream
        .map(new MergeTreeProjectionMapper<T>());
}
```

# Chapter 7
# 7 Evaluation

## 7.1 Evaluation Plan

We have selected the degree count, bipartition and triangle counting algorithms to evaluate with regards to various important aspects. We tested the algorithms in both single-node and distributed environments, using relatively large data sets as input. They were measured with different degrees of parallelism to test for scalability. Moreover, where applicable, we compared the up and downsides, as well as performance differences with the batch counterparts of these algorithms. This section gives detailed information on what particular tests were executed.

### 7.1.1 Sample Data Streams

The sample graphs we used for the bipartition and degree count algorithms with a local execution environment are the MovieLens data sets from GroupLens Research (2). These data sets represent users and their votes on movies. Since no two users or two movies can be connected, the graphs created using these data sets are clearly bipartite. The largest MovieLens data set contains 20 million ratings.

Since these graphs are all bipartite, they cannot be meaningfully used as an input for the triangle estimate algorithm. As this algorithm is rather slow, we simply generated random graphs with various sizes when testing in a local environment. Table 7.1 shows the main details of these graphs.

To generate the random graphs, we used the NetworkX (3) python package with a Barabási-Albert preferential attachment model (7). This results in random graphs that follow a power-law distribution, just like many real-world networks. This algorithm takes two parameters — the number of nodes and the number of edges to attach from a new node to existing ones. This latter parameter was set to a constant 30 for our sample graphs.

$$buffers = cores^2 \cdot machines \cdot 4$$

*Figure 7.1: Formula for Optimal Number of Flink Network Buffers*

The larger graphs used in the distributed experiments were from the SNAP collection (5). We used data from the Orkut social network (4), where each edge represents a friendship. This graph has more than 3 million vertices and above 117 million edges.

| Vertices | Edges | Triangles |
|---|---|---|
| 1000 | 19100 | 100431 |
| 2000 | 39100 | 143456 |
| 3000 | 59100 | 175196 |
| 4000 | 79100 | 194071 |
| 5000 | 99100 | 210812 |
| 6000 | 119100 | 226187 |
| 7000 | 139100 | 239208 |
| 8000 | 159100 | 253045 |

*Table 7.1: Graphs used for triangle count estimation*

### 7.1.2 Environment Setup

For the distributed tests, we used a Flink cluster with a job manager and 4 to 8 task managers. Each task manager had 8 task slots, which means the overall degree of parallelism was 64. Additionally, each task manager was permitted 16 gigabytes of heap space and 8192 network buffers, which define the network throughput when sending data to and from task managers. The formula to calculate the optimal number of network buffers is shown in figure 7.1 (1).

The large graphs were split manually into 8 pieces, so the job manager did not have to create the input splits before execution. This is a realistic scenario, because if we are not working from a file, each task manager could define its own streaming source, and then process it in parallel. Working from a file source was only considered for the experiments sake, to have a well-defined input for reproducible results.

## 7.2 Graph degrees

Our report only defines a single algorithm to count graph degrees and it also does not make sense to compare it with a batch version. Instead, we can test it for scalability

in both a local and distributed execution environment.

### 7.2.1 Single-node results

In the local execution environment we used the same MovieLens data set as for the bipartition experiments. To be able to compare the results to anything meaningful, we ran a "control" experiment as well, labeled *baseline performance*, where each edge is mapped to itself. This shows how long it takes to stream the whole data set itself, and then we can see how much overhead degree counting costs. We ran two versions of this degree count method — one where the parallelism is 4, and one centralized version where the parallelism is 1. This experiment was executed on a JVM with 512 megabytes of heap space.

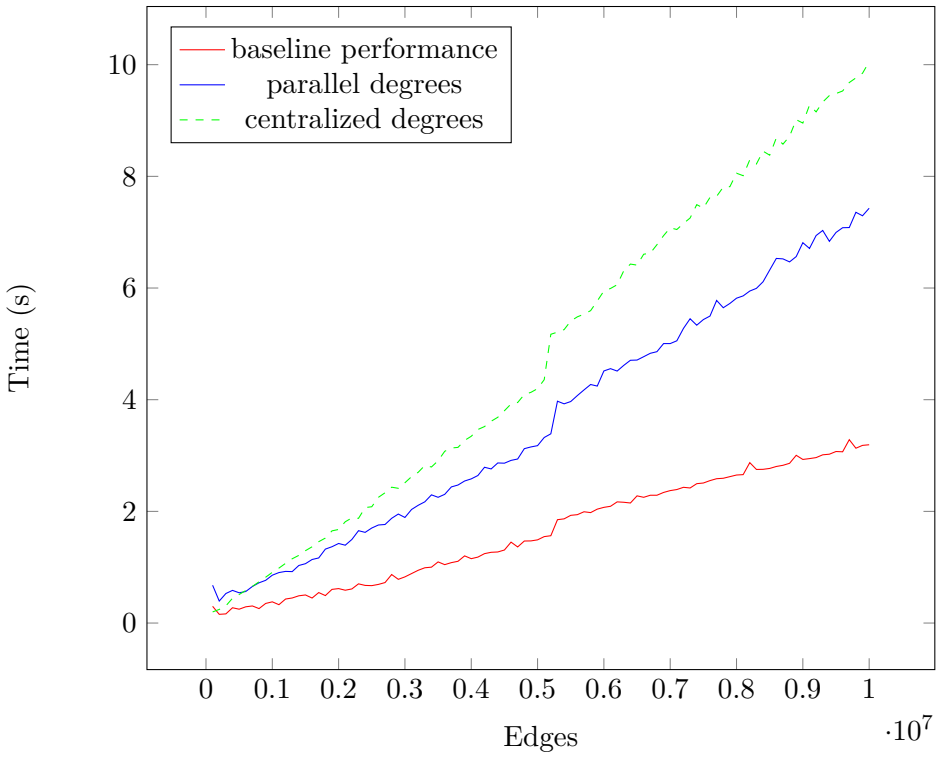



*Figure 7.2: Local Degree Count Results*

The streamed graphs were incremented by 100 000 edges at a time, with 10 million edges for the biggest locally tested graph. The results are shown in figure 7.2. The overhead caused by counting degrees seems to be linear with regards to the input graph size. Parallel execution is clearly faster than the centralized version here, as there is no overhead involved here, unlike in the bipartition algorithm. Each vertex is processed by a single sub-task, and no intermediate results need to be merged.

### 7.2.2 Distributed results

In the distributed environment, we used the same Orkut network graph as for bipartiteness check, using different sizes as the input for the algorithm. We executed this experiment on 4 nodes. The smallest graph we measured the degrees of contained 4 million edges, while the largest had 100 million. Each data point is the average of 3 executions. Figure 7.3 shows the results, compared to the baseline performance.

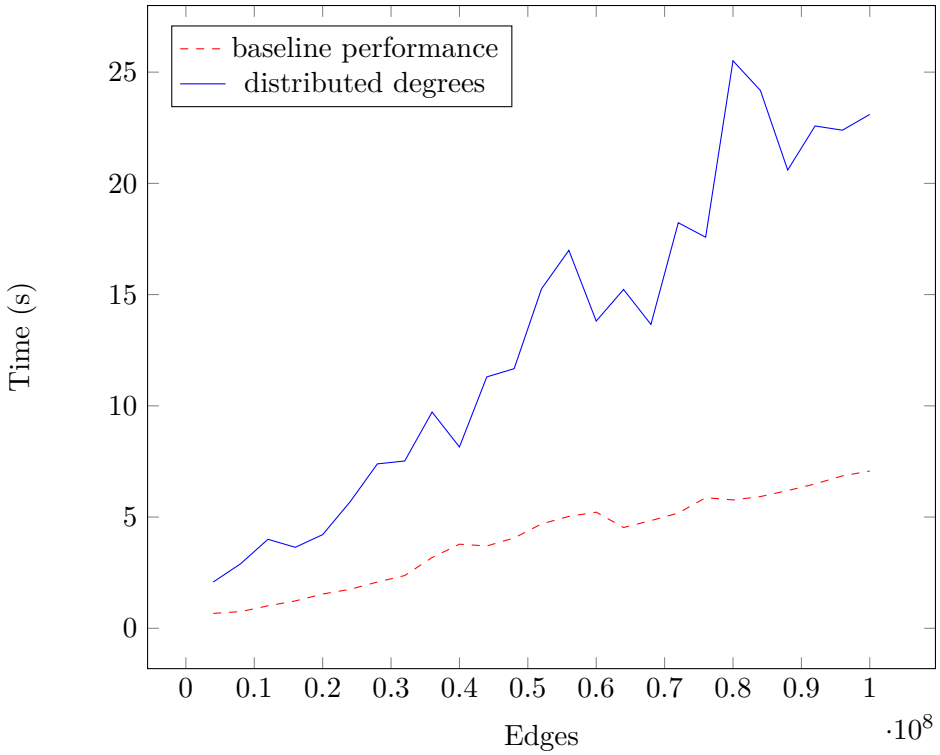



*Figure 7.3: Distributed Degree Count Results*

The results are not very stable with regards to execution time. This might be due to the usage of the cluster that the experiment was executed on. Nevertheless, the degree counting times show a linear trend.

## 7.3 Bipartition

For the semi-streaming bipartiteness check, we based our implementation on the first algorithm described in (10). We first created the centralized algorithm described in the paper, then made a distributed version, referred to as *aggregate* version in the following subsections. This simple distributed version was compared with the

windowed merge-tree structure, described in 4.2. Finally, the performance of the windowed merge-tree might change with different window sizes, so this is also an interesting experiment to run.

### 7.3.1 Single-node results

The graphs we used in the local environment were from the MovieLens data set, and rather small. As a result, we did not expect the merge-tree solution to greatly outperform the aggregate version, since the overhead of the merge-tree nearly outweighs the benefits of having a model that utilizes parallelism to a greater degree. It is nevertheless interesting to see the local results.

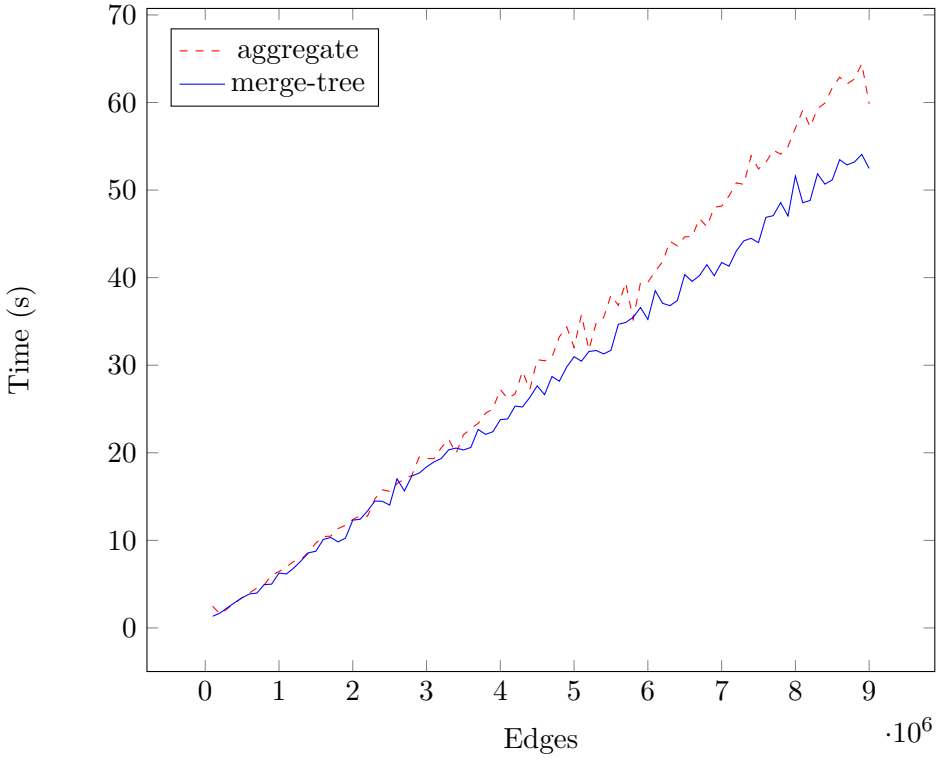



*Figure 7.4: Aggregating and Distributed Merge-Tree Bipartition Performances*

In this experiment, both solutions use the same parallelism, and both versions used windows of size 10 000. Figure 7.4 demonstrates the performance difference between the distributed version using the merge-tree and an aggregate version. The smallest graph in the experiment contained 100 000 edges, and the largest had 9 000 000. The merge-tree seems to outperform the aggregate version even on this smaller data set, although the difference is not large.

### 7.3.2 Different Window Sizes

We evaluated the performance differences between various sizes of windows. Three different window sizes were used: 1000, 10 000 and 100 000. Note that the size of each window is divided by 10 after every level of the merge-tree. The input graphs were generated by taking the first $n$ edges of the larger MovieLens data sets, with an increment of 100 000. The first test was executed with a parallelism of 4, and 512 megabytes of JVM memory space.

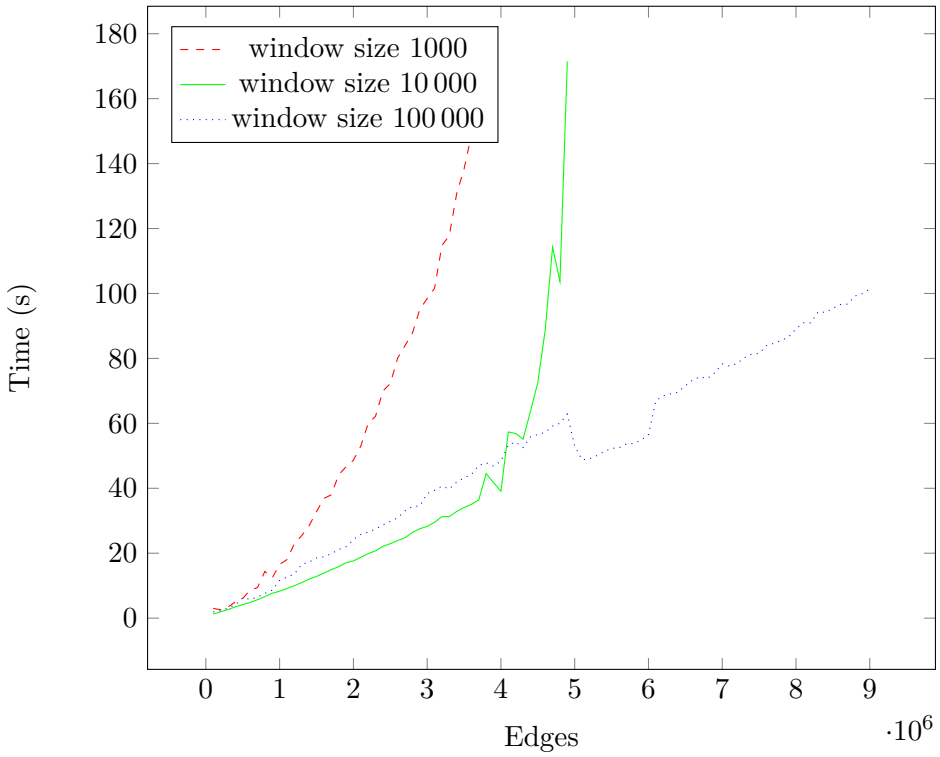



*Figure 7.5: Bipartition With Varying Window Sizes*

Figure 7.5 shows the results of this experiment. It is clear that a window size of 1000 is too small, and the communication overhead greatly reduces the performance of the algorithm. The JVM ran out of memory when trying to run bipartition on 3 800 000 edges with 1000 elements in the windows. With 10 000 elements in the window the JVM ran out of heap space on a graph of 5 000 000 edges.

After this, we ran another experiment on a single-node with 16 gigabytes of memory and 8 task slots. We also used larger window sizes, up to 10 million. Figure 7.6 shows our results. It is clear that windows of size 1000 are not scalable, but it is also visible that larger windows will cause more overhead.

These single-node results are certainly not conclusive, as the graphs are by no

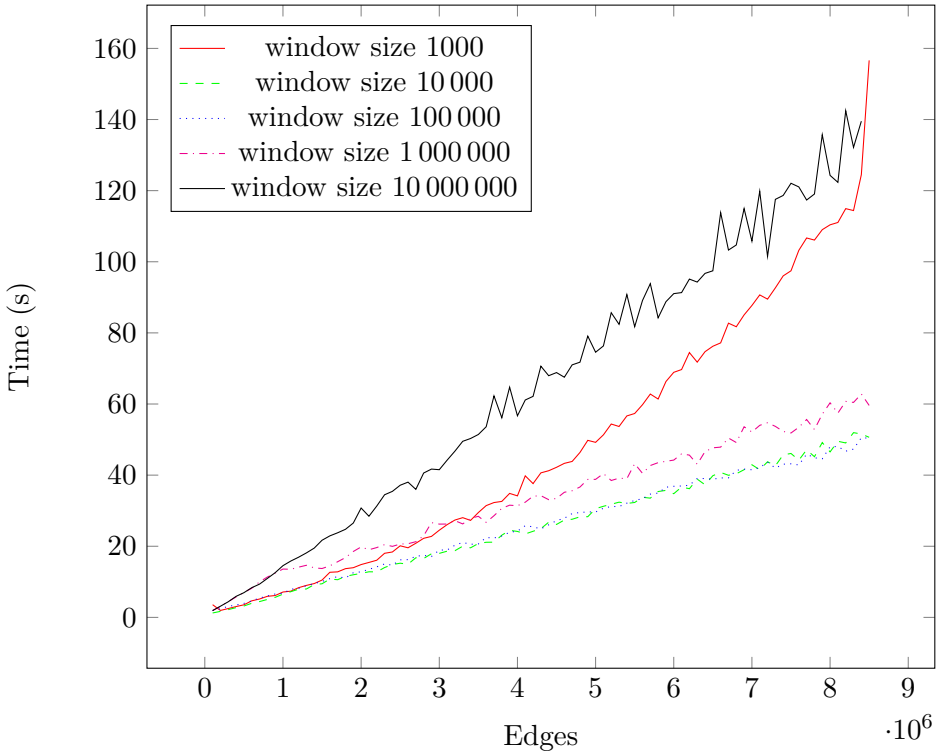



*Figure 7.6: Bipartition With Varying Window Sizes (2)*

means large. The distributed tests are a lot more important when evaluating this algorithm.

### 7.3.3 Distributed results

We executed bipartition check algorithm on the cluster described in 7.1.1, on the Orkut graph with more than 117 million edges. This is not a bipartite graph. The goal of the experiment was to show that this approach can spot a non-bipartite graph faster than a batch solution could, using the same algorithm. Three different tests were processed — the centralized version, where one mapper processes every edge, a distributed algorithm with a single, centralized aggregator, and finally, the distributed merge-tree.

| | First bad edge | First conflict (s) |
|---|---|---|
| Centralized | 35847 | 1149.52 |
| Aggregate | 26627 | 237.27 |
| Merge-tree | 12404 | 117.91 |

*Table 7.2: Distributed Bipartiteness Results on the Orkut data set*

| | First bad edge | First conflict (s) |
|---|---|---|
| Centralized | 40911 | 182.10 |
| Merge-tree | 66538 | 65.52 |

*Table 7.3: Distributed Bipartiteness Results on the MovieLens data set*

Table 7.2 shows the results. Each entry is an average of 3 executions. The second column shows the number of edges processed before an error was found. This value was counted per task, so it needs to be multiplied by 8 to get an estimate for the total number of edges processed before finding a conflict. While the centralized version is by far the slowest, the merge-tree solution produced the fastest results.

We also processed the MovieLens graph with 20 million edges in a distributed setting, comparing the merge-tree implementation to the centralized version on a smaller input. The original graphs were bipartite in this case, but we inserted a single edge that causes a conflict.

The results are shown in table 7.3. The merge-tree was faster in this experiment as well, but not as significantly as on the larger input. These results are the averages of 3 executions in this case as well.

It is important to note that the execution is expected to be much slower on this data set, since the original graph is bipartite, and only one edge breaks this property. While in the first experiment we could tell that a graph with 117 million edges is non-bipartite after 118 seconds, we needed 66 seconds for this second data set, which contains 20 million edges.

### 7.3.4 Comparing to Batch Environment

The centralized streaming version can be trivially converted to a batch algorithm, although this will not result in a well performing solution, as the two models require drastically different approaches when processing large data. As such, it would not be fair to compare the centralized streaming version to batch in this manner.

## 7.4 Triangle count estimates

As described in 5.1.2, we have two implementations for the same streaming triangle estimation algorithm. In this section we explore through our results and analysis whether the proposed improvement over the naïve implementation works in practice. The accuracy of the estimates is a secondary concern, since the original paper had suboptimal results for non-incidence streams, with error rates up to 40%, even with the

$$\sum_{i=1}^{s} \beta_i = \frac{\widetilde{T_3} \cdot s}{|E| \cdot (|V| - 2)}$$

*Figure 7.7: Relationship between the sample size, graph parameters and the beta values*

largest sample size. Another reason to ignore poor estimates is that our contribution is not merely the implementation of the algorithm — we were primarily interested in common themes that we can learn about during implementation and subsequently introduce to the graph streaming API. If the proposed incidence-sampling solution outperforms the "broadcast" version, we might indeed want to introduce such an operator to our API.

### 7.4.1 Single-node results

Even though we used graphs that follow a power-law distribution, degree distribution does not impact the speed of the execution of either algorithm. We use uniform sampling techniques, where samples have constant space and processing time while maintaining the same distribution. As such, the average number of edges that could connect either edge of the sampled edge to the sampled vertex — and hence need to be processed — will only be affected by the average vertex degree. This explains why the incidence-sampling solution is not influenced by degree distribution. For the simpler implementation it is clear that we process every edge a constant number of times regardless of degrees. The degree distribution of a graph does, however, have an impact on accuracy. The more triangles there are in the graph, the better we can estimate the total number of them.

The original algorithm assumes that we know the number of vertices and their identifiers in advance. Our algorithms require the same assumption, although the "broadcast" version can be executed without prior knowledge of vertices, with limited accuracy.

One key parameter of the triangle estimation algorithm is the number of sampling instances to run ($s$). Without prior knowledge of the size and degree distribution of the graph, it is impossible to guess a lower bound for this number, while still providing reasonably accurate results. By rearranging the equation in 4.6 we get 7.7, which provides an important constraint on $\beta$ and thus the $s$ value, which is as follows. Since we use $\beta$ to get an estimate for the number of triangles in the graph, if we want to have accurate results, this $\beta$ value should be large. This means that $T_3 \cdot s$ has to be several magnitudes larger than $|E| \cdot (|V| - 2)$. In conclusion, if $O(T_3) = O(|E| \cdot |V|)$ stands, we can assume $s$ to be a large constant value. Real-world graphs where the degree distribution shows a power-law seem to satisfy this constraint.

| | | Broadcast | | | Incidence | | |
|---|---|---|---|---|---|---|---|
| $\|V\|$ | $T_3$ | Estimate | Time (s) | Error % | Estimate | Time (s) | Error % |
| 1000 | 100431 | 100719 | 503.661 | 0.29 | 92933 | 134.809 | 7,47 |
| 2000 | 143456 | 152106 | 1058.408 | 6,03 | 143001 | 336.939 | 0,32 |
| 3000 | 175196 | 126878 | 1572.770 | 27,58 | 161898 | 543.084 | 7,59 |
| 4000 | 194071 | 218920 | 2308.748 | 12,80 | 175829 | 786.965 | 9,40 |
| 5000 | 210812 | 182813 | 2652.176 | 13,28 | 155164 | 985.129 | 26,40 |
| 6000 | 226187 | 199098 | 3239.577 | 11,98 | 236551 | 1372.265 | 4,58 |
| 7000 | 239208 | 175441 | 3793.170 | 26,66 | 305684 | 1654.211 | 27,79 |
| 8000 | 253045 | 229204 | 4286.067 | 9,42 | 229208 | 1881.138 | 9,42 |

*Table 7.4: Local triangle count results*

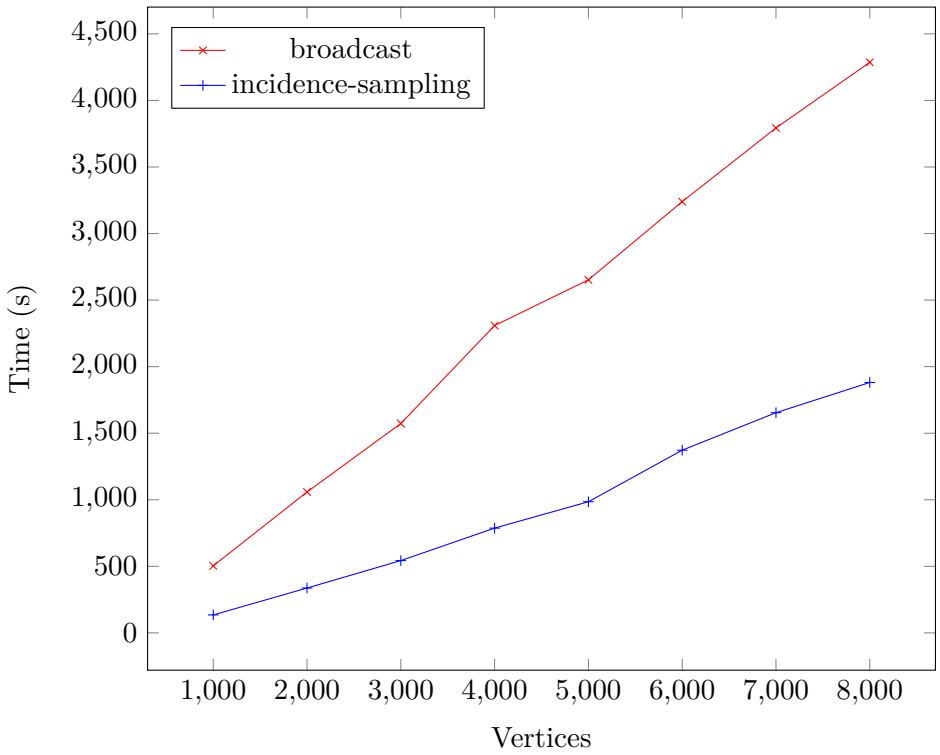



*Figure 7.8: Local triangle count speeds*

These tests were executed in a local environment, with a parallelism of 4, with 100 000 instances of sampling. The results of our two triangle estimation solutions are shown in table 7.4. The difference in the speeds of the two algorithms is plotted in figure 7.8.

We can see that the broadcast version is much slower than the incidence-sampling solution for these inputs, while the error rates of both algorithms are similar, and in accordance to what the original algorithm offers in terms of accuracy. In general

these errors are in lines with the results of the original paper.

The speed of both algorithms may seem very low, and even though it is by no means a fast solution, we have to consider that the number of sampling instances is quite high. This means that we have to process 159 100 edges 100 000 times in the case of the biggest graph with 8000 nodes, resulting in 15.91 billion rounds of the same sampling algorithm. The number of rounds processed in a second comes to 3.71 million in the broadcast version, while the throughput for the incidence-sampling solution is 8.46 million.

# Chapter 8
# 8 Conclusion

## 8.1 Discussion

To conclude this thesis report, we sum up our work and contributions in this section. First, we surveyed the state of the art research on graph streaming. After discussing different graph streaming models, our decision was to explore the simplest model first, in which only edges are streamed. As a result, the scope in this report is only the edge-only streaming model.

There are two key differences in our approach and current research. First, we always consider unbounded streams. This is a result of Flink's streaming model. This means that we cannot implement multiple-pass algorithms, because the first pass would never end, and currently structure loops or iterations that would support this are not implemented in the streaming programming model. The semi-streaming model, however, allows $polylog(n)$ passes, as well as that much stroage. Our model is therefore a restricted version of the semi-streaming one.

Secondly, if we want to exploit the distributed nature of Flink, we need to consider a distributed version of streaming algorithms that were defined in the related research. Even a seemingly simply convertible algorithm might end up causing far too much overhead, or a complete violation of the streaming principles. An example of this is the weighted matching example, where one solution would require to store edges in a queue and use locking.

With these differences in consideration, we implemented several graph streaming algorithms, such as global triangle count estimation, bipartiteness check, degree counts and weighted matchings. Where it was feasible, we created a distributed version as well.

Based on these implementations, we designed a graph streaming API, with commonly needed methods, to facilitate the creation of new graph streaming algorithms in a distributed setting. Besides the common functionality of our API, such as mapping and filtering edges or vertices, we propose 3 ways to aggregate intermediate results. These are the `aggregate`, `globalAggregate` and `mergeTree` methods. Finally, we describe

a custom way of sampling a stream of edges, based on reservoir sampling. This is not part of the API at present, but it could be a future addition.

The previously implemented algorithms were evaluated both in a local and a distributed setting. In some cases, multiple implementations were compared against each other. The distributed solutions were tested for scalability.

Our experiments show that the merge-tree structure is a faster way to process large data in distributed graph streaming algorithms, since it exploits the degree of parallelism to greater effect. Note that this is only the case with algorithms that store relatively complex state — algorithms such as degree counting would not benefit from this structure.

The triangle counting algorithm provides relatively accurate estimates and the incidence-sampling method seems to be faster than broadcasting all edges to every parallel sub-task. While it does not directly make sense to compare the performance of this algorithm to a batch solution, let us consider the use-cases where a streamed triangle estimation algorithm can be useful. This algorithm will work, and give reasonable estimates, even when the graph is huge, and it would not be viable to process with a batch algorithm. However, we will only get good estimates if the graph follows a power-law degree distribution.

In conclusion, we have created a new graph streaming model, utilizing continuous, unbounded edge-only graph streams. The API we have created for this model includes core functionality and various custom ways to merge results. This could be a great starting point for a fully developed graph streaming API, that supports different models as well. On top of that, we implemented previously researched graph streaming algorithms, then created distributed versions of them. These are part of our framework as examples.

## 8.2 Future Work

As previously mentioned, different graph streaming models may be interesting for different use-cases. CellIQ (12) is a real-time cellular network analytics system that uses a different model of graph streaming. In this model, graphs are formed when a large window is triggered, then processing takes place and the graph itself is discarded.

It would be possible to incorporate a similar concept into our graph streaming API. With a `snapshot` function a user could obtain a Gelly graph when a user-defined window is full. Furthermore this window could be a sliding window. In this case, we would need to update the Gelly graph, removing the old vertices/edges and adding the new ones.

# Bibliography


[1] Flink documentation: Configuring the network buffers. http://ci.apache.org/projects/flink/flink-docs-master/setup/config.html#configuring-the-network-buffers, 2015. Accessed: 2015-06-17.

[2] Movielens. http://grouplens.org/datasets/movielens/, 2015. Accessed: 2015-06-10.

[3] Networkx - barabási-albert graph. http://networkx.lanl.gov/reference/generated/networkx.generators.random_graphs.barabasi_albert_graph.html, 2015. Accessed: 2015-06-10.

[4] Orkut social network and ground-truth communities. https://snap.stanford.edu/data/com-Orkut.html, 2015. Accessed: 2015-06-17.

[5] Stanford large network dataset collection. http://snap.stanford.edu/data/, 2015. Accessed: 2015-06-10.

[6] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[7] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[8] Luca Becchetti, Paolo Boldi, Aristides Gionis, and Carlos Castillo. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *in KDD'08, 2008*, pages 16–24, 2008.

[9] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262. ACM, 2006.

[10] Joan Feigenbaum, Sampath Kannan, Andrew Mcgregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *In 31st International Colloquium on Automata, Languages and Programming*, pages 531–543, 2004.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[12] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. Celliq: Real-time cellular network analytics at scale. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[13] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[14] Andrew McGregor. Graph stream algorithms: A survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[16] Apache Spark. Lightning-fast cluster computing, 2013.

[17] Apache Storm. Storm, distributed and fault-tolerant real-time computation. 2014.

[18] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[19] Tiffani L Williams and Rebecca J Parsons. The heterogeneous bulk synchronous parallel model. In *Parallel and Distributed Processing*, pages 102–108. Springer, 2000.

[20] Jian Zhang. A survey on streaming algorithms for massive graphs, 2010.

# Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, June 30, 2015

.....

*János Dániel Bali*