DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

WinBro: A Window and Broadcast-based Parallel Streaming Graph Partitioning Framework for Apache Flink

ADRIAN ACKVA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

WinBro: A Window and Broadcast-based Parallel Streaming Graph Partitioning Framework for Apache Flink

ADRIAN ACKVA

Master in Computer Science
Date: August 6, 2019
Supervisor at KTH: Zainab Abbas
Supervisor at RISE: Ahmad Al-Shishtawy
Supervisor at RISE: Paris Carbone
Examiner: Prof. Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: RISE SICS

Abstract

The past years have shown an increasing demand to process data of various kinds and size in real-time. A common representation for many real-world scenarios is a graph, which shows relations between entities, such as users of social networks or pages on the Internet. These graphs increase in size over time and can easily exceed the capacity of single machines.
Graph partitioning is used to divide graphs into multiple subgraphs on different servers. Traditional partitioning techniques work in an offline manner where the whole graph is processed before partitioning. Due to the recently increased demand for real-time analysis, online partitioning algorithms have been introduced. They are able to partition a graph arriving as a stream, also referred to as a streaming graph, without any pre-processing step.
The goal of a good graph partitioning algorithm is to maintain the data locality and to balance partitions' load at the same time. Although different algorithms have proven to achieve both goals for real-world graphs, they often require to maintain a state. However, modern stream processing systems, such as Apache Flink, work with a shared-nothing architecture in a data-parallel manner. Therefore, they do not allow to exchange information along with parallel computations. These systems usually use Hash-based partitioning, that is a fast stateless technique but ignores the graph structure. Hence, it can lead to longer analysis times for streaming applications which could benefit from preserved structures.
This work aims to develop a state-sharing parallel streaming graph partitioner for Apache Flink, called **WinBro**, implementing well-performing partitioning algorithms. In order to do this, existing streaming graph algorithms are studied for possible implementation and then integrated into WinBro.
For validation, different experiments were made with real-world graphs. In these experiments, the partitioning quality, and partitioning speed were measured. Moreover, the performance of different streaming applications using WinBro was measured and compared with the default Hash-based partitioning method.
Results show that the new partitioner WinBro provides better partitioning quality in terms of data locality and also higher performance for applications with requirements for locality-based input data. Nonetheless, the Hash-based partitioner shows the highest throughput and better performance for data locality-agnostic streaming applications.

Sammanfattning

De senaste åren har det skett en ökande efterfrågan på att bearbeta data av olika sorter och storlek i realtid. En vanlig representation för många scenarier är diagram som visar relationer mellan enheter, till exempel användare av sociala nätverk eller sidor på Internet. Dessa grafers storlek ökar över tiden och kan enkelt överstiga kapaciteten hos individuella maskiner.
Grafpartitionering används för att dividera grafer i flera delgrafer på olika servrar. Traditionella partitioneringstekniker fungerar offline, där hela grafen bearbetas före partitionering. Baserat på den nyligen ökade efterfrågan på realtidsanalys har online-partitionsalgoritmer introducerats. De kan partitionera en graf som kommer strömmande, även kallad ett strömmande diagram, utan förbehandling.
Målet med en bra grafpartitioneringsalgoritm är att behålla datalokalitet och balansera partitionernas belastning samtidigt. Även om olika algoritmer har visat möjligheten att uppnå båda målen för realvärldsgrafik, behöver de ofta behålla ett tillstånd. Moderna strömbehandlingssystem, som Apache Flink, arbetar emellertid med en gemensam-ingenting-arkitektur på ett data-parallellt sätt. Därför tillåter de inte att utbyta information under parallella beräkningar. Dessa system brukar använda Hash-baserad partitionering, vilket är en snabb tillståndslös teknik men ignorerar grafstrukturen. Därför kan det leda till längre analystider för strömmande applikationer som kan dra nytta av bevarade strukturer.
Detta arbete har som mål till att utveckla en tillstånsdsdelande, parallellströmmande grafpartitionering för Apache Flink, kallad WinBro, som implementerar välpresterande partitioneringsalgoritmer. För att nå målet studeras befintliga strömmande grafalgoritmer för möjlig implementering och sedan integreras i WinBro.
För validering görs olika experiment med realvärldsgrafik. I våra experiment mäter vi partitioneringskvaliteten och partitioneringshastigheten. Dessutom kvantifierar vi prestanda för olika strömmande applikationer med WinBro och jämför den med en standard Hash-baserad partitionsmetod.
Resultat visar att den nya partitionern WinBro ger bättre partitioneringskvalitet när det gäller datalokalitet och även högre prestanda för applikationer med krav på lokalitetsbaserad inmatningsdata. Alternativt visar den Hash-baserade partitionen den högsta genomströmningen och bättre prestanda för datalokalitets-agnostiska strömmande applikationer.

Acknowledgments

I chose this thesis project because I wanted to be challenged with a system integration problem at the edge of research. Indeed, it was a challenge in some phases but the result is rewarding and makes me proud. But even more important is the fact that I learned a lot about research, programming, and Flink - and enjoyed it. Therefore, I would like to especially thank different persons for their guidance during my thesis journey.

Zainab, your profound knowledge about graph partitioning is impressive and I am glad that I could work with you. I especially enjoyed your confidence and trust in every situation of the project. This helped me to stay optimistic even when the development was stuck or external factors caused issues. I am looking forward to presenting WinBro at Flink Forward with you.

Ahmad, your experience and inspirations with regard to distributed systems helped me to work with Flink in a fully-distributed setup because I could rely on your hands-on skills inside and outside the server room. Moreover, I enjoyed your valuable contributions to our discussions about WinBro.

Paris, your insane Apache Flink knowledge and development experience was the third important factor for my contributions. You inspired me and Zainab to submit this thesis topic to Flink Forward and you despite your high workload you were always eager to answer questions or give advice.

Another big thank you goes to my examiner Vladimir Vlassov. It has been an honor to get such positive feedback from you who had worked longer in computer science and research than I have lived. I want to especially point out that I highly enjoyed your collaborative, solution-oriented style of examination.

Last but not least, I say thanks to my family and to Pascale. Throughout the first ideas about this Master program until the very last day, I received your support, could feel your love and trust in everything I do and did. *Danke*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last two years, 90 percent of all ever created data was produced [1]. Dealing with data of this size is a great challenge in information technology. Especially real-time analyses are increasingly important, for example, to detect credit card fraud or trends in social media. In this context, graphs play an important role because they can represent many real-world problems and scenarios. Graphs consist of vertices (nodes) and edges. While vertices usually represent objects or persons, edges connect vertices and symbolize relationships of any kind. For instance, the friendship graph of social network Facebook consists of over 1 billion users (vertices) and 140 billion friendships (edges) between them [2].
Analyzing real-world graphs of such size is not trivial because they are often too big to be processed or even stored on a single machine. Thus, partitioning techniques are used to divide graphs into multiple subgraphs on different servers. This task is especially challenging when graphs must be processed while they are still evolving, i.e. in real-time. This is also referred to as *streaming graphs* since their edges and vertices are continuously added or removed. Today, there exist different algorithms to partition these streaming graphs, such as HDRF or DBH.
However, stream processing systems which can process graphs, such as Apache Flink, are still missing a data-parallel partitioning algorithm providing good partitioning quality. Thus, this thesis aims to scale-out existing centralized partitioners and to develop a parallel streaming graph partitioner for Apache Flink.

1.1 Motivation

The combination of two different trends experienced increasing attention in industry and research: (a) Processing large graphs, and (b) streaming data analysis [3, 4]. When working with streaming graphs, applications have to flexibly handle graph data. It cannot be assumed how large, densely connected, or structured these graphs are. Thus, especially partitioning and processing large streaming graphs is a challenge.
Modern stream processing systems, such as Apache Flink, use Hash-based partitioning as a default technique to divide graphs to parallel working machines. Hashing allows assigning vertices or edges with their hash values very fast but ignores the graph structure. The consequence is a short partitioning period but a longer computation phase for the applications which rely on an intact graph structure. To address this problem, different algorithms for real-world networks were developed in the past, such as HDRF or DBH. They provide better partitioning quality in terms of low graph cuts because they maintain locality by keeping neighboring vertices in the same partitions. The downside of these algorithms is their need to maintain a state, e.g. a table with vertex degree information [3]. This is not complicated on a single machine. However, when the computation is done in parallel on large graph sets, this state information needs to be exchanged between machines.
Recent research has shown that there is an interest in partitioning graphs in a streaming environment. Furthermore, to partitioning graphs with billions of edges is generally feasible, as shown by [5]. Both single and parallel partitioners have already been integrated into stream processing systems, such as Apache Flink. But there is not yet a parallel partitioner available which provides good results in a shared-nothing environment.

1.2 Objective

Derived from the motivation, the goal of this thesis is to develop this streaming graph partitioner with data-parallel computations in a pure streaming environment framework, using existing partitioning algorithms. In detail, the following objectives are planned:

To study existing streaming graph partitioning algorithms and identify those which are suitable for unbounded streams while giving good results.

To study different state sharing techniques in Apache Flink and determine which of them can be used for efficient state sharing with streaming graph partitioning algorithms.

To develop a parallel streaming graph partitioner for a stream processing system with shared-nothing infrastructure

To evaluate the developed partitioner with experiments with real-world graph data sets for measuring the performance based on the following criteria: partitioning quality, partitioning speed and overall run time for graph applications, such as Connected Components Labeling or Bipartiteness Check. These results are compared with the performance of a regular Hash-based partitioner.

1.3 Contributions

This thesis comes with the following main contributions:

- A study of state of the art online graph partitioning algorithms with a focus on their usability for unbounded streaming graphs in a data-parallel context.

- The introduction of **WinBro**, a new parallel partitioning framework for the stream processing system Apache Flink. It is fully integrated with the Flink, and yields a partitioned *Data Stream* of *Edges* which can be used as input for applications in Flink. Its main advantage is that it computes partitions in a data-parallel manner along the stream pipeline, and does not need a shared state architecture or non-parallel operations for this.

- Parallel versions of two well-performing centralized online graph partitioners, namely HDRF and DBH, integrated into WinBro.

- An evaluation of WinBro's performance. Experiments in a fully-distributed Flink cluster show that WinBro delivers better partition quality than Hash on different real data sets, from a few million to 2.6 billion edges. Also integrated into streaming applications with a need for locality-based input data, the overall edge throughput is higher.

1.4 Limitations

Although this thesis has different contributions, some aspects can be considered shortcomings and delimit this work:

- The graph data sets and all experiments assume only growing graphs with an increasing number of edges, thus no edges are removed in the stream. Furthermore, every edge is streamed only once.

- All experiments were run on four physical servers so that any parallelism greater than four implied shared computing resources, i.e. no proportional scalability. Consequently, results based on higher parallelism are not representative in terms of performance.

- Both selected algorithms DBH and HDRF require state information to assign partitions. With regard to scalability, this limits the parallelism of WinBro to the available server resources because this state is replicated on all parallel instances.

- The two largest graph test sets (Twitter, Friendster) could not be partitioned with parallelism eight or higher. Furthermore, successful experiments with these two data sets only ran successfully twice, lowering the significance of the experiments.

- WinBro's network-intensive broadcast approach is practically bounded to local area networks because geographically distributed computations would cause high latency.

1.5 Methodology

This thesis uses different methods to meet the objectives presented in the previous section. A literature review is done to collect the state of the art in relevant. This includes 1) graphs, 2) streaming graph partitioning algorithms, and 3) stream processing systems. Based on this background information, the intended partitioner is designed and implemented. To finally validate the performance of this new partitioner, experiments are conducted, i.e. an empirical study is done.

1.6 Ethics and Sustainability

This thesis focuses on graph partitioning, a pre-step for data analysis. As written in the introduction, graphs can model various real-world scenarios. When persons are represented in graphs, these graphs can often contain sensitive information, thus privacy is very important. This concerns both legal and moral issues. A legal framework on how to collect and process data in the European Union is the *General Data Protection Regulation* which is in place since 2018 [6]. For this thesis, only anonymized data sets were used, so that graph vertices consist of numbers without indication to personal information.
Another aspect of data processing is security, implying how securely data is stored or transferred [7]. This issue is especially important when sensitive data is processed. Last, the environmental aspect is important for large data sets which are usually stored in data centers. According to Jones [8], their demand for electricity will reach around 7 percent of the overall electricity worldwide by 2030. Thus, it is important to reduce the consumption of resources here. The partitioner presented in this thesis is designed to decrease the run time of streaming graph applications. Hence, it helps to increase the efficiency of these applications and finally saves energy.

1.7 Structure of the Thesis

The thesis first provides relevant theoretical concepts in the background chapter, covering graphs, graph streams, and stream processing systems including Apache Flink. Related Work presents the state of the art in the area of graph partitioning concepts and algorithms. Chapter 3 presents the concept and design of WinBro, followed by Chapter 4 which describes the implementation of WinBro into Apache Flink
The experimental setup with explanations and metrics is elaborated in Chapter 5, followed by Chapter 6 where the results of these experiments are presented and discussed. In the end, Chapter 7 closes the thesis with Conclusion and Future Work.

Chapter 2

Background

This chapter presents the theoretical background of important concepts to create a basis for all subsequent chapters. First, graphs are introduced in general, followed by data and graph streams and stream processing systems including an overview of Apache Flink. Thereafter, the state of the art in streaming graph partitioning and related work are presented in Sections 2.5 and 2.8.

2.1 Introduction to Graphs

The research area of graph theory is a well-studied field in mathematics and applied to several research areas, such as biology, social science or physics. This section briefs about the most important graph-related areas of this thesis. A graph G consists of two disjoint sets, a vertex set *V(X)* of size *n* and an edge set *E(X)* of size *m*. When two vertices are directly linked, an edge is formed. These vertices are often referred to as source vertex and target vertex, respectively. Thus, they are adjacent to each other, or simply called *neighbors*. When graphs are illustrated, it is common to use diagrams [9], as in Figure 2.1. It shows a **simple graph** with four vertices, connected with four edges. It is called simple because it has neither vertices looping to themselves nor multiple edges between two vertices Newman [10].

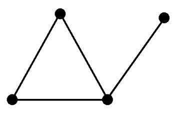


Figure 2.1: Simple undirected graph

In addition to simple connections, Bondy, Murty, et al. [11] show that graphs can be directed or undirected. While the graph above is undirected, it is also possible to indicate orientation with arrows, i.e. pointing from source to target vertex. Directed graphs are common in different application areas, but also used for Page Rank, a metric to compute the importance of a website, introduced by Google [10]. However, this thesis only deals with undirected graphs. Thus, this topic is not further elaborated.

Centrality

Newman [10] defines centrality as "how important vertices (or edges) are in a networked system". There exist different interpretations of centrality in graph theory, for instance, Eigenvector centrality or Betweenness Centrality. Among these different metrics, **Degree Centrality** is one which can be calculated with ease. The **degree** of a vertex is defined as "the number of edges attached to it" [10]. When setting this in relation to the total number of vertices in a graph, the result is **normalized degree centrality**.

Subgraph

A graph can be divided into multiple subgraphs, where edges of the subgraph are also edges of their supergraph. Thus, when parts of a graph are separated from each other, different subgraphs exist. This is important for both graph partitioning and parallel graph streams. Assuming that subgraphs do not change when being partitioning or processed, they can be reconstructed to the initial super graph. Formally, Godsil and Royle [9] define a **subgraph** Y of graph X as shown in Equation 2.1:

$$V(Y) \subseteq V(X), E(Y) \subseteq E(X) \tag{2.1}$$

Power-Law Graph

Many real-world graphs follow a near power-law degree distribution [10] [12]. When plotting the degrees in a logarithmic scale, they form a slope, as visible in Figure 2.2 from Twitter social network. This representation shows that only a very small number of nodes n has a high centrality $C(n)$ whereas the majority has only a few neighbors, i.e. followers. These kinds of networks are also known as "scale-free networks" [10], and very common in social networks, but also in many other areas. However, this topic has some controversy because many graphs are statistically not fully *scale-free* as this paper from Klarreich

[13] investigates. Nonetheless, for this work's research, this mathematical discussion is not too relevant. Section 2.8 will show that partitioning power-law graphs is a special challenge and how this is achieved.

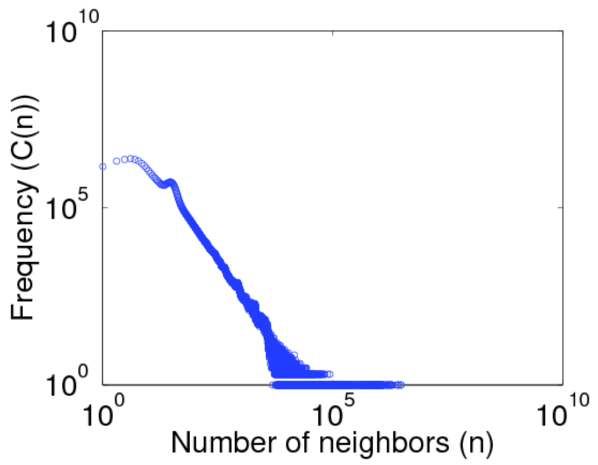


Figure 2.2: Power-Law Degree Distribution of Twitter [14]

The metric to determine whether a graph follows power-law degree distribution is the constant **power-law exponent** α. Equation 2.2 shows the probability of a vertex to have a certain degree d. When $\alpha \to 0$, the density of a graph is high and the degree distribution is not skewed. The opposite happens when $\alpha \to \infty$, where only a few vertices have the majority of all connections. Then these vertices can also be called **hubs**. In fact, the power-law exponent of most real-world graphs ranges between 2 and 3. [10]

$$P(d) = d^{-\alpha} \qquad 2.2$$

2.2 Data Streams and Graph Streaming

This section introduces to graph streams and considerations when processing them. But since graph streams are technically data streams, this concept is introduced first. Section 2.3 covers different stream processing frameworks which are used to work with such streaming graphs.

2.2.1 Data Streams

In the context of data stream processing, the term stream refers to that elements are continuously sent record-by-record, potentially unbounded. One stream element can be simple (e.g. a number or a word) but can also be more complex, e.g. a Facebook post including its metadata. It is possible to process them only once, known as *single pass*, or multiple times. No matter how often stream records arrive, one challenge of data stream processing is how to store and aggregate information. Due to data streams' unbounded nature, servers can quickly reach their storage limits, so that saving the whole stream is not advisable. Instead, different techniques exist to get insights into streaming information.

A **synopsis** does not store every element but maintains a state of one or more desired metrics, for example, a maximum value of a stream. Thus, every element is inspected whether this condition is met and the variable is adjusted accordingly.

Windows can be used as an alternative or complement to synopses. A window can be considered as "temporary buffer [...] to split an unbounded data stream into a smaller batch of tuples" as defined by Sajjad et al. [15], where a tuple is one stream record.

The scope of a window can be different:

A common implementation is **time-based** with a distinction between *tumbling* and *sliding* windows. Tumbling windows have a fixed time without overlapping stream elements whereas sliding windows also have a fixed time but can be overlapped. A simple example for a tumbling window is to compute an average for <u>every</u> 60 minutes. In contrast, a sliding window computes a new average every ten minutes over the <u>last</u> 60 minutes.

Fixed sized windows evict elements once their buffer is full, e.g. 50 elements.

2.2.2 Graph Streams

A graph stream consists of a potentially unbounded number of edges or vertices arriving one-by-one in a continuous manner. It is possible to represent streaming graphs in three different ways:

Edge-only streams consist of edges, arriving in any order.

Combined Vertices and Edges streams have two different streams, one of vertices, one of edges.

Triplet streams carrying source and destination vertex as well as a value assigned to them, such as the weight or other indicators.

The order of graph steams can be random but two different models are usual - **Incidence Model** and **Adjacency Model** [10]. In case of an incident-based model, a vertex arrives with all its connected edges, i.e. its neighbors, at the same time. In a pure streaming context, this approach is unsuitable because the number of edges and their connections is not known in advance. Thus, the Adjacency model more common. Here, edges arrive in any order without further dependencies. Nevertheless, storing the whole graph can quickly exceed the processing servers' capacities, similar to data streams. For graphs, the *Semi-Streaming Model* [16] attempts to lower the space required for graph streams by exploiting the observation that most graphs have a significantly lower number of vertices than edges. Hence, storing vertices only reduces the demand for memory.

2.3 Stream Processing Systems

Before moving to the graph partitioning section, this section provides an overview of different stream processing systems which are capable of handling stream graphs. It is also justified why the choice of a processing system for this implementation is Apache Flink. Moreover, Flink is presented in more detail with a focus on streaming and graph partitioning as well as state management.

2.3.1 Stream Processing Frameworks

With the rise of big data and the need to compute new data with latency after its creation, several stream processing systems have evolved. Stepping back to the year 2008, Dean and Ghemawat [17] introduced the **MapReduce** programming framework which enables run programs in parallel on large clusters with commodity machines. Its two main operators *map* and *reduce* first distribute subtasks of bigger analysis jobs to parallel worker nodes, and second to combine them again and the end. As a result, the overall computation time is reduced. Hadoop [18], the most popular framework of this kind, was the basis for the following tools. Apache Spark [19] makes calculations faster with in-memory data storage instead of writing intermediate results to disks. However, both systems process batches of data and are not suitable for real data

streams which process records one-by-one. Consequently, the developers of Apache Flink, Carbone et al. [20], introduced a real streaming framework in 2015 which is also compatible with batch processing.
Today, there exist more alternatives, both proprietary and open-source, but Flink and Spark are among the most famous and established ones. In the past, several performance benchmarks between Flink and Spark were performed [21] [22] [23], giving different results. This thesis, however, does not aim to conclude which framework works fast but which one is more suitable for (graph) stream processing. Thus, the processing technique is relevant. Apache Spark processes elements in batches and when referring to streaming, it processes them in *micro-batches* [24]. Consequently, records are not streamed one by one but in (very) small groups. Flink instead streams records internally record-by-record. Thus, it is a pure stream processing system and is used for the implementation in the next chapter.

2.4 Apache Flink

Apache Flink is an open-source data processing framework which allows both batch but especially stream processing. Figure 2.3 shows the Apache Flink Stack with Flink's different layers. Starting from the bottom, Flink can be run in different environments, such as local, distributed and also in a (public) Cloud setup. Its core is the runtime with the *Distributed Streaming Dataflow* which is explained later in this section. For batch processing exists a *DataSet API* whereas data streams are processed with the *DataStream API*. These application programming interfaces (API) are enriched with different libraries, for example for tables or machine learning but also *Gelly* for graph processing. Unless differently marked, this chapter is based on Carbone et al. [20] and the official documentation [25]. Since this work is about streaming, all described concepts refer to the Streaming API or are generally applicable to Flink.

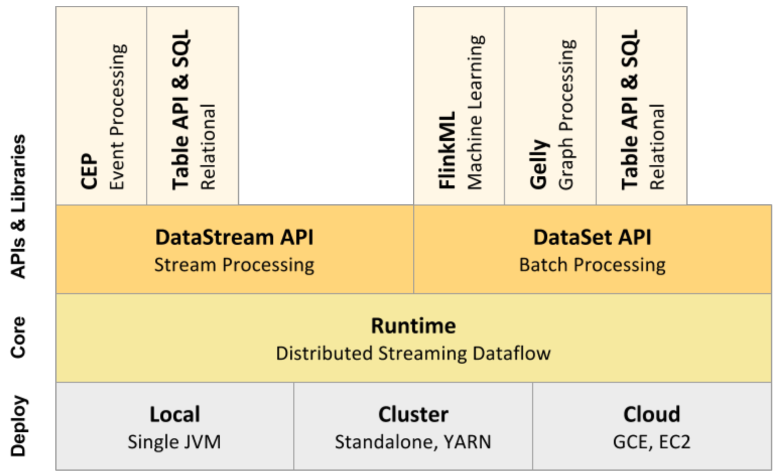


Figure 2.3: Apache Flink Stack [26]

2.4.1 Dataflow Graph & Job Execution

A Flink program, usually referred to as *Flink Job*, is commonly displayed as **Dataflow Graph** in the form a directed acyclic graph (DAG) containing one or more *sources*, transformations (*operators*), and one or more *sinks*. A sample graph is presented in Figure 2.4. Source functions can read from files or other sources. Typical transformations instead use the input data and compute according to the instructions. When all calculations are completed, stream elements are sent to the sink operator.

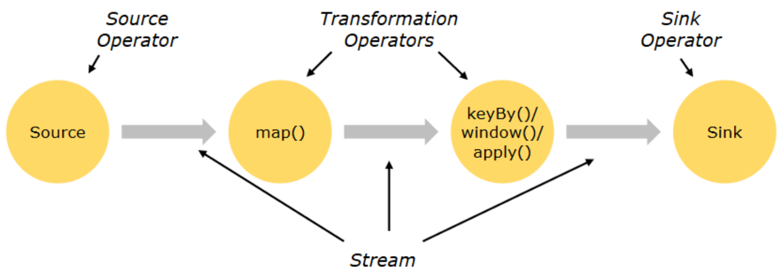


Figure 2.4: Sample Dataflow Graph [27]

Most of the job operators can be run in parallel. This is where coordination is required, solved by Flink with a master-workers architecture. The master is called **Job Manager** and the workers are **Task Managers**. Every task manager is a Java Virtual Machine (JVM) and performs tasks and subtasks assigned to it. A task is created based on the operators from the Dataflow Graph. resulting in parallel, distributed execution of the Flink job.

2.4.2 Windowing Streams

An important feature of data streams in Flink are windows. To fully exploit their functionality, different prerequisites are required, such as setting a notion of time and a keyed stream. Both concepts are explained in the following, before Flink's implementation of windows is further elaborated.

Notion of Time

When processing data streams in Flink, different time notions can be used for different purposes. They are presented below:

Event Time: When it is important to consider the actual moment when an event happened, the event time notion is used in Flink. So, when data comes with a timestamp, this can be extracted and Flink processes the record in accordance with the actual event. For instance, when sensor data arrives with a certain latency but the original time of the measurement matter, event-time is the preferable choice.

Ingestion Time: In contrast to event-time, ingestion time is the moment when a data record is registered in Flink after arrival.

Processing Time: The last notion of time is processing time. In this case, the current processing time of Flink matters. This method can be used without synchronization among task managers and is fast but can lead to out of order processing of records which were subsequent in the source system. Depending on the application, this can be disadvantageous.

KeyBy Function

By keying a stream, Flink has an entry point on how to distribute streams across task managers for different operators in the beginning. To key a stream, a user-defined function can be written or different pre-defined options can be chosen from. Keyed streams are also important for windows, as shown in the next section.

Windows

As presented in Section 2.2 windows are an important technique for data stream processing. Consequently, Flink takes advantages of them, too. All windows require a keyed data input if parallel task execution is desired.
The window types in Flink are tumbling, sliding, session and global. While the first three types were presented earlier, the global session is briefly described: All windows with the same key are assigned to the same window. This approach enables custom *trigger* and *eviction* functions. With these functions, it is possible to create custom rules or policies.
Once items of a window are evicted, a *process* function is used to perform certain actions. Without detailed explanations, it is possible to use them for **reduce**, **fold**, or **aggregate** functions. Moreover, custom **process** functions can be defined.

2.4.3 Sharing State

Flink offers various ways of sharing state along the dataflow graph. This is an important feature of Apache Flink because all tasks are executed strictly in parallel and there is no shared state across them unless a state sharing mechanism is configured. Thus, there is no communication between parallel-running operators even if the task managers are located on the same physical machine.
Generally, one can distinguish between **data-parallel** and **task-parallel** execution. A data-parallel execution strategy allows to partition data to task managers and to concurrently perform the same calculations on this partitioned data. Task-parallelism refers to simultaneous computations of different tasks and operators on the same or on different data sets. [28]
The strategies to share state are different and range in their complexity. Besides simple data forwarding between operators along the pipeline, it is possible to broadcast state and send it to all other task managers working with the same operator. Furthermore, the state can be stored per key or operator on a local task manager. Moreover, different backend state solutions are provided by Flink to handle very large state or to ensure fault-tolerance. [28]

2.5 Graph Partitioning

The rising amount of data combined with higher computational power comes to a challenge: big graphs often exceed single machines capacities' to process or store them. An often-cited research paper by Stanton and Kliot [4] points out important aspects of graph partitioning:

NP-Hardness: It is an NP-hard problem to achieve optimal load balance and minimize the number of cuts in the graph structure. NP-hard describes the impossibility to solve a problem that satisfies both conditions [29]. Consequently, existing partitioning algorithms attempt to either perform well on one of these two extremes, or at least to find a compromise.

Locality: Locality in real-world graphs results usually results in good "natural partitioning". Hence, it is preferable to keeping clusters intact by assigning them to the same partition. By cutting edges or vertices in a purely random way, communities are destroyed with high probability.

Cost: Communication cost on machine level is lower than communication across servers. Even in local area networks, "latency is measured in microseconds while inter-process communication is measured in nanoseconds" [4]. Thus, communication overhead across machines over the network should be as low as possible.

In addition to these aspects, two other general paradigms are important for graph partitioning. The first one refers to whether to partition *edges* or *vertices*. The second one is about *online* and *offline partitioning*. They are presented in the following two subsections.

2.5.1 Online and Offline Partitioning

Depending on the environmental setup and use case, graphs can be partitioned offline and online. Offline partitioning is a more traditional approach and assumes that the graph data set is fully available before partitioning process starts. For instance, the total number of edges and vertices is known and identifiable. Many frameworks and algorithms, such as METIS [30] take this information as an input for their computation. In stark contrast to this, online partitioning assumes that graph data continuously arrives in the target system and must be partitioned incrementally. In today's world, streaming graphs are very common but their behavior cannot be predicted. Thus, when edges and vertices arrive, they might change the structure of a graph. For example, when observing trending topics on Twitter, an unexpected event could become popular and would have an impact on the overall "Like" and "Retweet" structure. Another aspect of streaming graphs is that the total graph size is unknown and

potentially unlimited because events and likes occur in the future.
A side effect of little graph knowledge in online partitioning is that a **state** must be maintained. This may be degree information, vertex or edge assignment or partitioning loads. Managing state is less relevant for offline partitioning because it can use knowledge about the graph and also iterate over graphs for better results, whereas online partitioning focuses on a one-pass strategy due to the streaming setup. Summarized, online partitioning deals with several uncertainties that offline partitioning does not have.
Table 2.1 shows the characteristics of both techniques, based on [4, 3]. Especially implications of online partitioning characteristics are relevant for Related Work and Implementation.

Table 2.1: Offline and Online Graph Partitioning

Characteristic	**Offline**	**Online**
Graph knowledge	full and a-priory	incremental, no prior
Partitioning based on	full graph	subgraph & aggregations
State management	less relevant with to prior knowledge	maintain with communication overhead
Development paradigm	METIS framework	focus on 1-pass algorithms

2.5.2 Vertex & Edge Partitioning

Since a graph consists of edges and vertices, two partitioning approaches are popular in research: Vertex partitioning and edge partitioning [12]. This section presents both techniques.
Vertex partitioning splits a graph into multiple subgraphs and distributes its vertices to k partitions. In order to achieve this, vertex partitioning cuts edges between vertices where necessary. This is the reason why this technique is alternatively called *edge-cut partitioning*. Figure 2.5 provides a small example with a graph divided into two partitions.
Edge partitioning also splits a graph into multiple subgraphs but sends its edges to k partitions. Hence, edges remain intact but vertices are cut. Therefore, another name for this method is *vertex-cut partitioning*. In fact, vertices are replicated when an algorithm decides to cut them. Hence, all subgraphs can be re-connected when all copied vertices are identified. Figure 2.6 illustrates this technique.

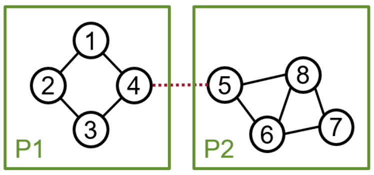


Figure 2.5: Vertex Partitioning

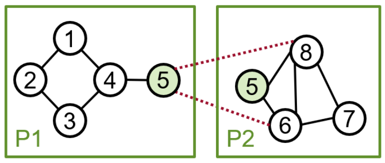


Figure 2.6: Edge Partitioning

Quality Metrics for Edge Partitioning

When recapping the goals of graph partitioning, the number of cuts should be low and all partitions should be equally balanced. Applying these two objectives to edge partitioners, two metrics are common:
The first indicator is the **replication factor**. It is the fraction of the number of replicated vertices over the number of vertices in total, as shown in Equation 2.3. Hence, if the value is small, only a few copies are needed to partition all edges. If the replication factor is higher, many vertices were split across partitions.

$$\sigma = \frac{Total\ Number\ of\ Vertex\ Copies}{Total\ Number\ of\ Vertices} \tag{2.3}$$

The second quality metric for edge partitioners is the **load balance** of edges in all partitions. The more equal the total number of edges placed into partitions, the better is the load. The goal should be a load balance close or equal to 1 where all partitions hold the same number of edges. Equation 2.4 provides the exact calculation:

$$\lambda = \frac{Number\ of\ Edges\ on\ Highest\ Loaded\ Partition}{\frac{Total\ Number\ of\ Edges}{Number\ of\ Partitions}} \tag{2.4}$$

To measuring the quality of vertex partitioners, the load balance is calculated in the same way, except that vertices are counted instead of edges. However, the cutting metric is expressed in edge cuts where the total number of edge cuts in divided by the total number of edges. Since this thesis does not implement vertex partitioners, this is not further elaborated.

2.6 Graph Partitioning Algorithms

After having studied the different forms of graph partitioning, this section provides an overview of existing algorithms and briefly describes the advantages and disadvantages of using them for streaming graphs at the end.

In an experimental study by Abbas et al. [3] from 2018, a comprehensive comparison shows different algorithms for streaming graph partitioning, optimized for both vertex and edge cuts, as well as a generally applicable Hash algorithm. As implied by the goals of graph partitioning, these algorithms aim for low cuts and a good balance. An exception is Hash which concentrates on getting a good balance.

2.6.1 Vertex Partitioning Algorithms

Linear Deterministic Greedy (LDG) [4] is a vertex partitioning algorithm with the objective to maximize the number of neighbor vertices in one partition. Thus, the edge-cut ratio should be as low as possible. In order to do this, LDG calculates the neighbors of all vertices and keeps track of the partitions in which vertices are located. Thus, for every vertex, every partition is searched for most common neighbors. However, LDG sets a capacity limit per partition based on the total number of vertices and edges, which must be known before the algorithm starts.

Fennel [31] is a proprietary algorithm developed by Microsoft. Its idea is based on LDG but using another scoring mechanism and introduces different parameters to tune the partitioning result. With this optimization Fennel often outperforms LDG [31] [3]. However, it still requires information about the graph before starting the process.

2.6.2 Edge Partitioning Algorithms

Greedy [12] is an edge partitioning algorithm assigning edges according to their vertices' historic assignments. Thus, it maintains a state of all vertices and the partitions they are placed into. This approach aims to have a low vertex cut, i.e. a low replication factor. When an edge arrives, it follows these rules:

1. If both vertices are already the same partitions, assign them to the least-loaded common partition

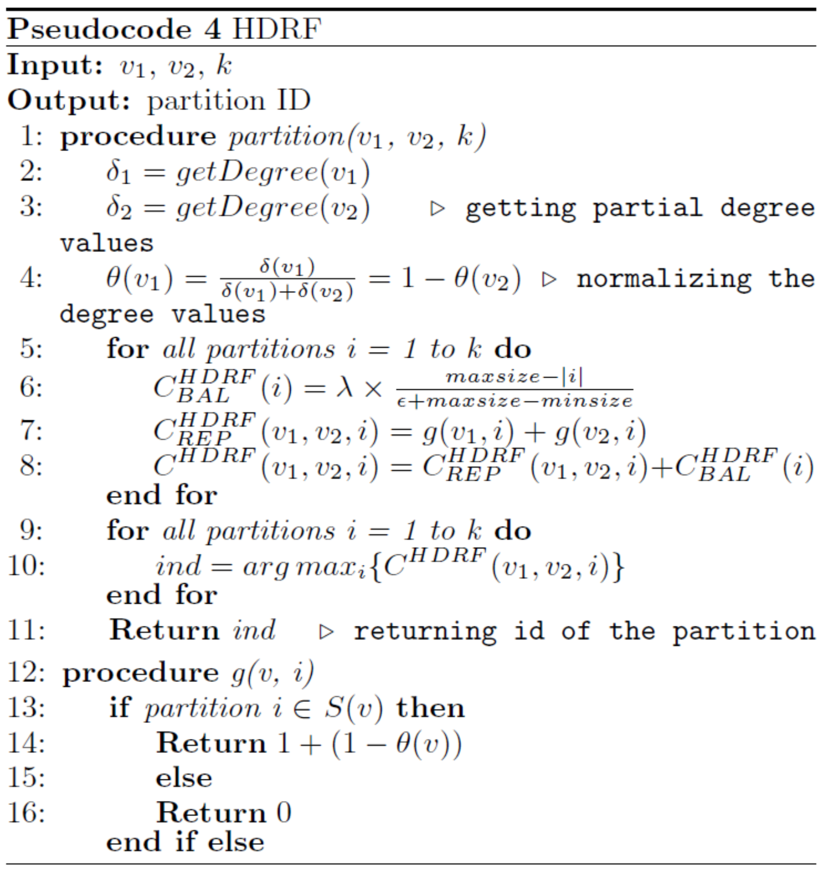
2. If both vertices are known, assign to the least-loaded of the already assigned partitions

3. If only one vertex is new, assign the edge to the least-leaded partition of the already assigned partitions of that one

4. If both vertices are new, assign the edge to the least-loaded partition

Greedy does not need any information about the graph before it starts, but it requires to store all vertices, their partitions and loads of all partitions.

High-Degree Replicated First (HDRF) [5] has been particularly developed to process power-law graphs. The motivation is that most of the real-world graphs have a small number of hubs and a large number of low-degree vertices, as pointed at in Section 2.1. As the name suggests, the aim of HDRF is to replicate high-degree nodes across partitions. This should lead to a low replication factor with fewer vertex cuts.
To find the optimal partition for an edge, this algorithm chooses the highest *HDRF score* among all partitions when an edge arrives. This score is calculated with the aid of 1) the resulting replication factor when this edge is placed there, and 2) the load balance of the partitions. For a good cut ratio, this calculation takes degree information of both vertices and their previously assigned partitions into account. In addition, parameter λinfluences the partitioning load. With the right setting, it handles the load imbalance when the order the stream elements is known before. Otherwise, when using a default value of $\lambda= 1$, it is more agnostic to order. Further implications of changing $\lambda$are explained in [5].
Although the evaluation of HDRF follows later in this chapter, the pseudo code is shown below. This is because HDRF is one of the three algorithms which are used for the implementation project in this thesis.



```
Pseudocode 4 HDRF
Input: $v_1$, $v_2$, $k$
Output: partition ID
 1: procedure $partition(v_1,\ v_2,\ k)$
 2:     $\delta_1 = getDegree(v_1)$
 3:     $\delta_2 = getDegree(v_2)$     ▷ getting partial degree
    values
 4:     $\theta(v_1) = \frac{\delta(v_1)}{\delta(v_1)+\delta(v_2)} = 1 - \theta(v_2)$  ▷ normalizing the
    degree values
 5:     for $all\ partitions\ i = 1\ to\ k$ do
 6:         $C^{HDRF}_{BAL}(i) = \lambda \times \frac{maxsize-|i|}{\epsilon+maxsize-minsize}$
 7:         $C^{HDRF}_{REP}(v_1, v_2, i) = g(v_1, i) + g(v_2, i)$
 8:         $C^{HDRF}(v_1, v_2, i) = C^{HDRF}_{REP}(v_1, v_2, i) + C^{HDRF}_{BAL}(i)$
        end for
 9:     for $all\ partitions\ i = 1\ to\ k$ do
10:         $ind = arg\,max_i\{C^{HDRF}(v_1, v_2, i)\}$
        end for
11:     Return $ind$    ▷ returning id of the partition
12: procedure $g(v,\ i)$
13:     if $partition\ i \in S(v)$ then
14:         Return $1 + (1 - \theta(v))$
15:         else
16:         Return $0$
        end if else
```

Figure 2.7: Pseudo Code of HDRF (Abbas et al. [3])

Degree-based Hashing (DBH) [32] combines partial degree information and hashing to choose a partition for an arriving edge. It looks up the degrees of both vertices, compares them and hashes the edge to a partition, based on the value of the vertex with the lower degree. This makes DBH suitable for power-law graphs without prior knowledge about them. However, it requires maintaining a degree counter state for every vertex of the stream. [3]
Because DBH is also used to implement the project of this thesis, the pseudo code for it is printed below:

```
Pseudocode 5 DBH
Input: $v_1$, $v_2$, $k$
Output: partition ID
1: procedure $partition(v_1, v_2, k)$
2:     $\delta_1 = getDegree(v_1)$
3:     $\delta_2 = getDegree(v_2)$
4:     if $\delta_1 < \delta_2$ then
5:         Return $Hash(v_1)mod(k)$
6:         else
7:         Return $Hash(v_2)mod(k)$
       end if else
```

Figure 2.8: Pseudo Code of DBH (Abbas et al. [3])

The last partitioning algorithm presented in this thesis is **Grid** [33] which creates a squared map, also named grid, of partitions. Arriving edges are hashed to one of the fields in the grid. From there, all partitions with the same row or column index are considered potential destination partitions. Out of this set, the partition with the lowest load is selected for the edge.
Grid comes with the limitation that the number of partitions that must be represented in a squared matrix, i.e. it must be divisible by four. Moreover, Grid needs to store the load of all partitions but does not need information about the graph in advance.[3]

2.7 State in Streaming Graph Partitioning

All algorithms presented in the previous sections have in common that they require a state to partition streaming graphs online. This was also pointed out in Section 2.5.1. Bearing the objective of this thesis in mind (data-parallel streaming graph partitioner without shared state), the topic **state** requires closer examination. Table 2.2 shows the state requirements for all presented algorithms from the previous section. Similar to the algorithm overviews, it is based on Abbas et al. [3]. Since METIS is an offline partitioning framework, it is not part of this list. Hash does not require state, as mentioned earlier. Besides this, all algorithms store all distinct vertices of a graph. Furthermore, all but DBH also keep track of the assigned partitions. HDRF, the best performing, also needs degree information, similar to DBH. Consequently, all algorithms require at least a state as big as its number vertices.

Table 2.2: State requirements of selected partitioning algorithms

Algorithm	**Information stored in the state**
LDG	vertices, partition assignment
Fennel	vertices, partition assignment
HDRF	vertices, degree, partition assignment
DBH	vertices, degree
Grid	vertices, partition assignment
METIS	prior state (offline)
Hash	–

Looking at existing solutions from graph processing systems, the following alternatives to maintain state or exchange information are possible:

1. **Shared-memory or shared-disk**: If a system's architecture allows this approach, all data-parallel pipelines can perform read and write operations on shared memory or shared disk. Examples for graph processing framework with a shared-memory approach are X-Stream [34] or Ligra [35]. Flink takes advantage of a shared disk for its backend state, too. However, this solution aims to enable fault-tolerance rather than state exchange between nodes [28].

2. **Message passing:** This or other direct communication approaches use mechanisms to exchange information iteratively. With regard to distributed graph processing systems, Pregel [36] uses such a mechanism but works iteratively with so-called *supersteps* for every round. In a streaming graph environment, this method is not suitable, as discussed earlier.

3. **Local State (Replication)** is an alternative as long as the size does not exceed the capacity of a server. Here, the state is locally stored in-memory or on a disk, similar to the locally executed program. The advantage is a full picture of all state information, but the drawback is the required space. Apache Flink provides different stateful functions and mechanisms such as *Operator State* or *Keyed State*, as described in Section 2.4.2.

4. **External applications** like databases or message brokers such as Kafka can also store information. If this approach is chosen, the external tool needs to be available during the whole partitioning process and be able to cope with numerous read and write operations in parallel. 2.4.2.

In general, it is possible to use any of the above-listed state management methods. However, modern stream processing systems, such as Apache Flink or Apache Spark, use a shared-nothing architecture by default because they run data-parallel. The consequence is that they cannot use shared-memory. Furthermore, their programming model does not intend message passing in a way that Pregel does. Since Pregel is also no stream processing system, it is no alternative for this work. Moreover, an external tool is also not suitable because the goal is a streaming graph partitioner fully integrated into Apache Flink. Hence, a connection to another tool is complicated because it might require additional setup.
The variety of mechanisms suggests that there is no optimal way of how to work with a state because all options have advantages and drawbacks. At the time of this thesis, all well-studied and well-performing streaming graph partitioners require state, at least as big as the number of vertices. Since this work aims is to provide a data-parallel streaming graph partitioner in a shared-nothing setup, this constraint is taken into account and different possibilities to limit the state size are targeted.

2.8 Related Work

The goal of this thesis, to my best knowledge, is the first attempt to integrate parallel versions of well-studied online graph partitioning algorithms into a stream processing system without a shared state. This aims to improve the speed of streaming graph applications by using these algorithms instead of the default Hash-based method. Therefore, this section focuses on partitioning algorithms that can be a base, but also a benchmark, for the implementation project of this thesis.
Verma et al. [37] compare different partitioning algorithms and their performance for distributed graph processing systems, such as PowerGraph, PowerLyra or GraphX. Among others, algorithms presented in the previous section are used for this. However, these graph processing systems are no real stream processing systems. Thus, they are not suitable for this work.
When it comes to offline graph partitioning, METIS [30] is a well known and often referenced framework. Different papers for offline and online graph par-

titioning use METIS computations to benchmark partitioning quality. However, this framework and all other offline partitioning algorithms need information about the data set in advance, mainly the number of edges and vertices. These prerequisites make them unsuitable for unbounded streams where no assumptions about the structure can be made.
The same applies to LDG [4] and Fennel [31] algorithm for vertex partitioning. Although they were tested and validated in a streaming environment and referred to as streaming algorithms, their input graph parameters need to be known beforehand.
Concerning streaming graph edge partitioners, all four presented algorithms (Greedy, HDRF, DBH, Grid) could possibly process unbounded streams. The study from Abbas et al. [3] shows that the best overall performance among these algorithms can be observed at HDRF. It provides very good cuts and it gives satisfying load balance results. These results make HDRF a suitable candidate. Since DBH also requires degree information, it is also used for the implementation.
When concerning state, an important aspect of parallel computations, there is no clearly outstanding example among these four alternatives. Though, Hash has no state requirements and is the default option to partition data in stream processing systems.
Table 2.3 shows both algorithms which are identified for the implementation. Hash is also added for comparison.

Table 2.3: Characteristics of Selected Partitioning Algorithms

Algorithm	**Optimization**	**State Requirements**
HDRF	Cuts	All degrees, assigned partitions, partition loads
DBH	Cuts	All degrees
Hash	Speed/Load	No State

Chapter 3

Data-parallel Streaming Graph Partitioner

This chapter is dedicated to the conceptual design of **WinBro**, a parallel streaming edge graph partitioner. The actual implementation into Apache Flink is described in Chapter *Implementation*.
Based on the finding from Related Work in Section 2.8, the overall best performing edge partitioning algorithm, HDRF, is chosen for WinBro. Alternatively, WinBro can be used with the parallel version of DBH.

3.1 State-based Parallel Partitioning

Two approaches can be considered to implement state-based parallel partitioning. One approach is to use a single **global state** shared between parallel partitioning tasks. The alternative is to maintain **local states** in the parallel partitioning tasks. Applied to stateful partitioning algorithms such as DBH or HDRF, they come with different advantages and drawbacks.
Figure 3.1 illustrates the edge flow from reading edges until their assignment to a partition using a global state. For example, in the case of DBH and HDRF, degree counting is performed in parallel, and all the degree information is aggregated into the global state in the second step. This single operator possesses the ground truth with all information about the edges. However, it is also a bottleneck, causing lower throughput since the global state operator cannot process edges with the same pace as they arrive. Furthermore, the whole idea of parallelism becomes obsolete because the flow merges at one point.

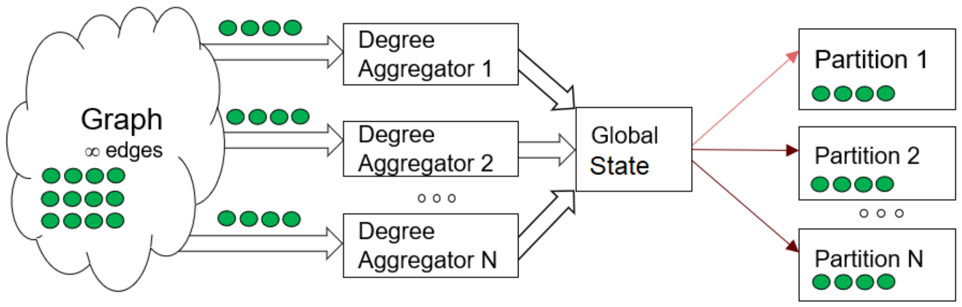


Figure 3.1: Global State Design

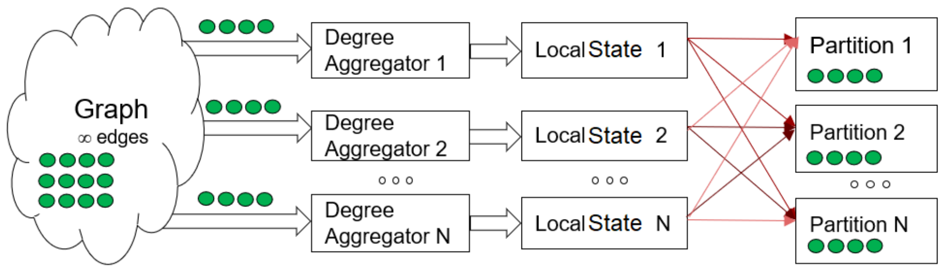


Figure 3.2: Local State Design

An alternative is the usage of local states, as shown in Figure 3.2. Here, it is possible that all states work independently. This approach is fast and works without a bottleneck. However, there is no communication between local states or parallel partitioning tasks, and the knowledge is limited to the subgraph processed in each parallel stream. Since HDRF and DBH require full information about degrees to work well, the missing holistic view is likely to cause partition assignments based on incomplete information. Table 3.1 summarizes the findings of the global and local state approach.

Table 3.1: Global and Local State compared for WinBro

Characteristic	**Global State**	**Local State**
View	whole graph	local subgraph
Synchronization	across nodes	—
Parallelism	single bottleneck	high

Since the two previous approaches are not optimal, WinBro combines both designs to benefit from certain of the above-mentioned advantages. For this purpose, state exchange with broadcasts is an essential part of WinBro. Thus, it is possible to get a global view without a bottleneck. Specifically, by adding this functionality, it is possible to send degree information from one task manager to all other task managers. This guarantees that every local state can work independently but receives all vertex degree information to make profound partitioning decisions. As with all state sharing techniques across machines, this solution comes with a network overhead. But servers are presumably located in the same local area network so that no significant delay is expected.

3.2 WinBro Programming Flow

This section is dedicated to the actual implementation of WinBro. It includes an overview of the whole DAG, which operators it uses, and how windows and broadcast are integrated into this setup.

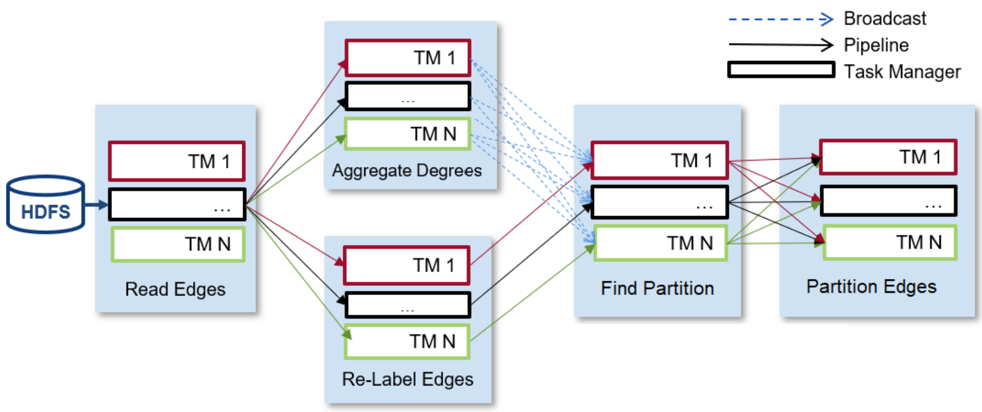


Figure 3.3: WinBro DAG

3.2.1 Job Graph Design

WinBro's DAG is illustrated in Figure 3.3 and is generally adjustable to different parallel edge partitioning algorithms requiring shared state. In total, five operators are used to partition a streaming graph in parallel. Please note that reading is done with a parallelism of 1 but all other steps run in parallel. To provide a brief overview, they are shortly described below. Detailed presentations follow in the next sections.

1. **Read Edges:** In the beginning, the input data is read from text file or HDFS and distributed to all task managers. Additionally, all edges are labeled with a unique identifier (ID). This process is explained in Section 3.2.4.

2. **Aggregate Degrees:** Based on all incoming edges, this operator counts the degrees of all vertices. The result of this phase is 1) a vertex degree table containing degree information of all vertices assigned to this window, and 2) a unique window identifier. They are combined into a tuple and broadcasted to all other instances after every window.

3. **Re-Label Edges:** On the other side of the fork, the input edge IDs are relabeled in this operator with the aid of windowing. This is a crucial step to perform a join operation in the next phase where degree information and edges are merged.

4. **Find Partition:** All partitioning logic happens in this operator. In asynchronous order, it processes arriving edges and broadcasts. These broadcast inputs contribute to a local in-memory state for all vertices and degrees that grows over time. Inside the same operator, all edges are partitioned with (modified) DBH and HDRF algorithms. The result is a tuple with an edge and the chosen partition ID.

5. **Partition Edges:** The final step is to use a *partitionCustom* method to send the edge to its destination partition. Alternatively, this can be used as input for applications.

Keying and Windowing the Graph Input Stream

WinBro works with **windows** to process smaller amounts of edges at a time. Thus, there is also a need to identify a reasonable key applicable to all edges. Based on the background provided earlier, the following decisions are made for WinBro:

Keying by source vertex is a pragmatic way for WinBro because no prior knowledge about the graph structure can be assumed. This decision accepts the risk that many edges with the same source vertex at the same time, the load might be unbalanced.

Tumbling windows ensure no overlapping and require no conditions. Hence, they are more suitable than sliding or policy-based windows.

3.2.2 Aggregate Degrees

DBH and HDRF require vertex degree information of all arrived edges. Hence, Aggregate Degrees is the first phase of WinBro. Figure 3.4 illustrates the degree aggregation with a small example. For simplicity, parallelism is not considered here. Given that all edges are in the same time window, those with the same source vertex are grouped together. These three groups (pentagons) are forwarded one-by-one to the Aggregate Degrees operator. The result is one vertex-degree table per key and window, tagged with a window ID, different for every key. Every *Output* is individually broadcasted to all other task managers.

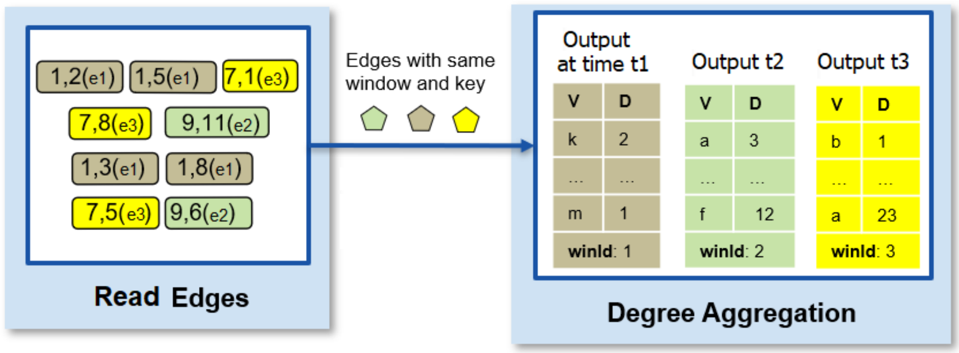


Figure 3.4: Aggregate Degrees in WinBro

3.2.3 Re-Label Edges

The Edge Re-labeling operator processes the same keyed and windowed edge stream as the Aggregate Degrees, i.e. all edges are in the same windows. This allows identifying whether an equivalent broadcast degree table has already arrived at the Partitioning operator. For better understanding, the concept of window ID assignment is presented in Figure 3.5. All edges in the same window are looped over and the window identifier is formed. Afterward, the edges are emitted to the next operator with this window ID, replacing the edge ID.

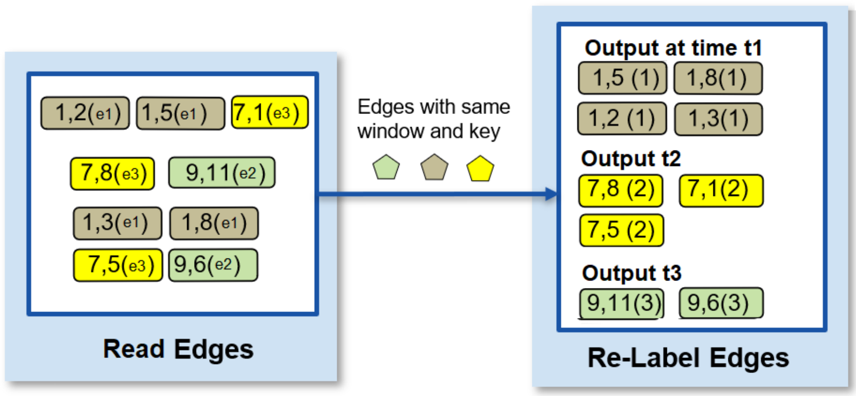


Figure 3.5: Re-Label Edges with a Window ID

3.2.4 Partitioner Match Function

The heart of WinBro is the Partitioner Match Function operator which has two main tasks:

1. Combine and merge all arriving broadcasts into a local state with a global view, so that every task manager has a full picture of all vertex degrees processed by WinBro.

2. Choose the destination partition for all arriving edges based on this snapshot of a global view and the given algorithm.

Due to the complexity of the Match Function, it is described in different sections in this work. The overall procedure concerning the integration of the broadcasts is explained in this section whereas the technical details are given in the Implementation chapter. Furthermore, the parallel versions of DBH and HDRF algorithms are presented in Section 3.3.1 and 3.3.2.

Before any edge can be assigned to a final partition, all partial degree maps are integrated into the local vertex degree map. Since there is no order guarantee for edges and broadcasts in different stream processing systems, there is a requirement to ensure that the degree map is updated before the algorithm assigns a partition ID. This is explained in the next section.

Join Edges and Broadcasts with a Window ID

A problem occurs when an edge arrives before its equivalent degree information from the broadcast is integrated into the global state. WinBro uses a custom **Window ID** approach to ensure this. The window ID is based on the individual keyed edges and their IDs and it is guaranteed that both Aggregate Degrees and Re-Label Edges calculate the same window ID independently. This window ID allows finding join broadcasts and edges in the Match Function.

All broadcasts (and all window IDs) arrive at every node in the stream pipeline. Also, every window of edges eventually arrives at one node once. Consequently, all edges will eventually see their counterpart broadcast. So, even without order guarantees, all Window IDs will have a match on one of the task managers. WinBro takes advantage of this and creates temporary window ID entries for all edges and broadcasts. Once all edges and broadcasts with the same identifier arrive, it is ensured that the broadcast information is integrated into the global state and the edges can get a partition assignment. All details about the selection of this method and its alternatives are presented in Appendix A1.

The joining process is as follows: When a new broadcast element or a new edge arrives, it checks if its window ID exists in a join table. If this is not the case, it creates an entry and either adds an edge or sets the size (broadcast) for this new entry. This is repeated for every edge in this window. Calculating the number of edges of an entry with a broadcast element is done by summing up all degrees in the partial degree table and dividing the result by two. If a broadcast arrives late, the size is updated later. A join table entry is complete when the entry size is equal to the number of edges arrived. Then, it is added to a list of completed joins. After all edges from the window ID entry have partitions assigned, the entry is removed from the join table and also released from the *Complete List*. This ensures that the entries are stored temporarily only.

To give a visual example of this process, Figure 3.6 displays the final state of a partitioner match function. Task manager *TM 1* shows its full picture whereas the others only show extracts. For simplification, every window ID has the same color. Broadcasts elements are symbolized with squares and edges with circles. For every *ready* Window ID entry, the edges have assigned partition IDs. They are sent to the last operator, presented in the next section.

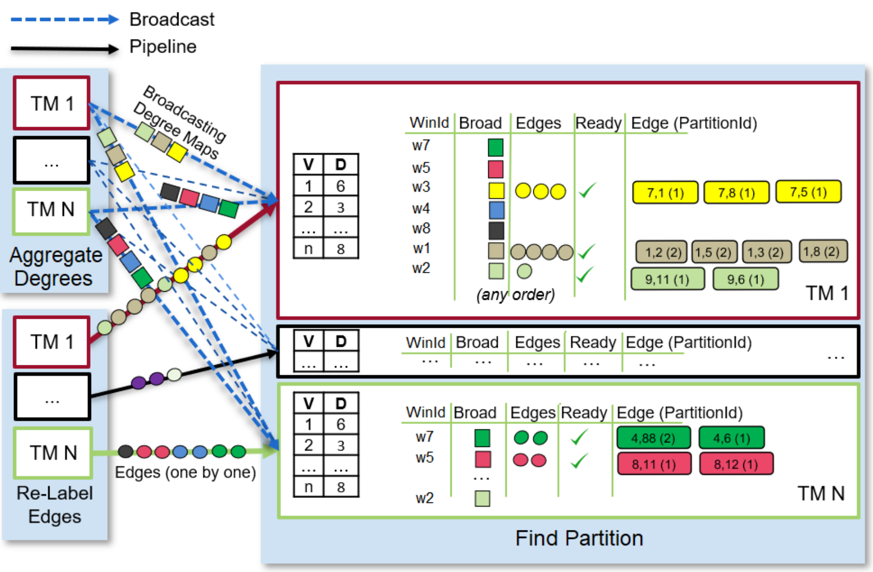


Figure 3.6: Joining and Partitioning in Match Function

Although this method passed the tests, it comes with certain overhead. In addition to the global vertex degree map, a join table needs to be maintained, too. This increases the need for memory. Furthermore, the number of broadcasts per tasks manager is always higher than the number of edge windows, as visible above. The consequences are described in Section, 4.3.

3.2.5 Partition Edges

In the last step of WinBro's partitioning flow, all edges are partitioned with a *partitionCustom* operator which get the partition ID as the parameter which then sends the edge it the selected partition. This last step is illustrated in Figure 3.7.

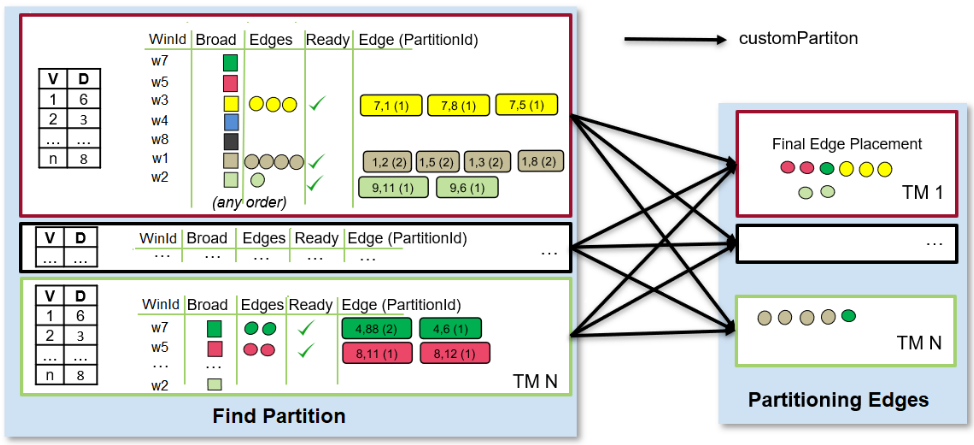


Figure 3.7: Partition Edges in WinBro

3.3 Algorithm Adjustments for Parallel Processing

3.3.1 HDRF

Pure HDRF, as presented in Section 2.6, is not designed to run in concurrently in a shared-nothing environment. Hence, an analysis during the design phase led to two changes to create a parallel version of HDRF from the original one: First, edge counting can be done in advance without synchronization. Thus, it is suitable for parallel computation. This adjustment does not require a big change and can be achieved by simply not increasing the degree of vertices during the partition selection. Instead, the algorithm sees the (until that point) intermediate degree when looking it up. All other calculations, such as the score or the machine load require a full picture of the state. As mentioned at the beginning of this chapter, reconciliation is not part of WinBro in its current stage. Thus, neither scores nor vertex partitions are synchronized across task managers.

The second change does not touch parallel computations but is used to cope with the reduced state, as shown in the next chapter. One consequence of this design decision is that after a cleanup some low degree vertices might not present in the global degree map anymore when HDRF selects a partition for a new edge. In this case, a temporary vertex with degree 1 is created to partition the edge. This does not significantly affect partitioning quality because the

cleanup methods only remove low degree vertices.
The code to select a partition with the parallel version of HDRF can be found in Appendix A3 where all differences compared to the original HDRF code are marked. The implementation of the algorithm is based on HDRF partitioner in the repository created for Abbas et al. [3]. Besides the marked changes, the underlying state classes are also adjusted, for example, to allow ignoring missing edges after state cleanup or for Reservoir Sampling integration.

3.3.2 DBH

Adjusting DBH towards a parallel execution has the same consequences as modifying HDRF: The vertex degree count is done in the degree aggregation operator in a previous phase and a placeholder edge temporarily supports the partitioner, if the original vertex has been removed. Since these changes are similar to HDRF changes, they are not further explained here.
The code for DBH in its parallel version can also be found in Appendix A2 where also all changes are marked. Similar to the original version of HDRF, the code for the DBH partitioner from Abbas et al. [3] is reused and accordingly adjusted for WinBro.

Chapter 4

Implementation

After having described the architecture and general design of **WinBro**, this chapter presents the implementation into Apache Flink. It follows the same structure as the previous chapter, i.e. presents the overall structure and the individual operators subsequently.

4.1 WinBro Dataflow Graph

WinBro's Flink DAG was presented in Figure 3.3 in the previous chapter, and Code 1 shows the procedure of WinBro at a high level. In the following subsections, the implementation of these five operators is described.

Read Edges

Read edges outputs a *SimpleEdgeStream* from Flink Gelly library [38]. Its edges contain both source and target vertex but also a value that can be used for weights, for example. Besides edges, it comes with a streaming context and automatically assigns timestamps to incoming edges. WinBro takes advantage of both edge value and timestamps for windowing and edge ID creation respectively. Since edges are consumed in text format in the first place, the text files are transformed into an Edge instance. This requires a *flatMap* operation.
Technically, this operator is not strictly parallel. For a real parallel setup, custom Flink functions need to be implemented, or external tool (e.g. Kafka) are required [39].

Algorithm 1 WinBro Dataflow

```
// Read Edges from File
SimpleEdgeStream<Edge> edgeInput = readTextFile(input)
     .flatMap(String -> Edge<source, target, id>)

// Aggregate Degrees
BroadcastStream<Tuple<Degree Table, WindowId>>
   degreeAggregate = edgeInput
      .keyBy(source)
      .timeWindow(seconds)
      .process(new DegreeAggregator)
      .broadcast()

// Re-label Edges
DataStream<Edge> edgesWindowed = edgeInput
      .keyBy(source)
      .timeWindow(seconds)
      .process(new EdgeWindowId)

MatchFunction matchFunction = new MatchFunction()

// Match Function
DataStream<Edge> partitionedEdges = edgesWindowed
      .keyBy(source)
      .connect(broadcastStream)
      .process(matchFunction)

// Partition Edges
      .partitionCustom(PartitionId)
      .addToSink()
```

Aggregate Degrees

The degree aggregation operator is used to broadcast partial degree information to all other task managers. It is implemented with Apache Flink's **Broadcast State Pattern**. This state sharing technique was initially added to Flink to exchange information between parallel executions. It uses a broadcast to send data to downstream operators so that all task managers receive them and can work with this information. To join a broadcast and a regular stream, it is necessary to combine them with this broadcast state pattern [40]. Flink requires windows to allow broadcasts in the dataflow. Hence, the edge input stream is keyed and windowed before the degree aggregation starts. In this context, it is important to mention that this class is newly instantiated for every window and every key because every windowis processed individually and only partial degree information is broadcasted. In other words, whenever a process function is called, it creates a new degree table for a (little) subgraph of the whole graph. Code 2 shows how this count is performed.

Algorithm 2 Aggregate Degrees class procedure

```
  input: all edges in one window

  procedure createDegreeMap {

     for (both vertices v of all edges in window) {
        if (degreeMap contains vertex)
           degreeMap.vertex(v).setDegree += 1;
        else
           add vertex to degreeMap, degree = 1;
     }

  output: Tuple(degreeMap, WinodowID)
 }
```

Re-Label Edges

In order to have a matching edge for every broadcast element, this operator creates the unqiue window ID. It uses the same mechanism as Aggregate Degrees but emits only edges and forwards them to the next operator.

Find Partition

The fourth operator chooses the partition ID for all arriving edges. It connects and co-processes the degree aggregator stream and the re-labeled edge stream. In contrast to the Aggregate Degrees and Re-Label Edge operators, the Match Function class is instantiated once per Flink job, not per window. Thus, every update remains in the degree map and can be accessed during the whole partitioning process.
Code 3 provides a simplified overview of the Match Function class. Every time when a broadcast arrives at this operator, **processBroadcastElement** is

called. In contrast, an arriving edge triggers the **processsElement** to find a partition for this edge. These two functions are described below.

Algorithm 3 High Level Methods of Partitioner Match Function

```
void processBroadcastElement(degreeMap,windowId)
   updateGlobalDegreeMap(degreeMap);
   checkIfEdgesArrived(windowId)
   choosePartitionId()

void processElement(Edge edge)
   checkIfBroadcastArrived(edge.getWindowId)
   choosePartitonId()
```

1. **ProcessBroadcastElement:** A broadcasted degree map is first integrated into the global state, i.e. it is checked whether the vertex ID already exists. If so, it adds the degree count from the local state. Otherwise, it adds the vertex to the global state with the according degree. The other two methods, *checkIfEdgeArrived* and *choosePartitionId* are necessary to enable the window join operation and to emit all edges respectively. The latter is called in both process functions because the missing order guarantee of Flink can lead to broadcasts arriving after edges. If this was the case without a method call in the broadcast section, edges might be several waiting edges at the end.

2. **ProcessElement:** All arriving edges are processed here. Due to the necessity to join by window IDs, this calls *checkIfBroadcastArrived* and finally emits all edges marked "complete".

All three functions, *checkIfEdgeArrived*, *checkIfBroadcastArrived* and *chooseP-artitionId* are shown in Code 4.

Algorithm 4 Find Partition with Window ID Join

```
// called by processBroadcastElement()
void checkIfEdgesArrived()
      if (windowId in joinTable)
         set size of windowId entry
         if (sizeBroadcast == sizeEdges)
            add entry to completeList
      else
         add windowID to joinTable

// called by processElement()
void checkIfBroadcastArrived()
      if (windowId in joinTable)
         add edge to windowId entry
         if (sizeBroadcast == sizeEdges)
            add entry to completeList
      else
         add windowId to joinTable

// called by both process functions
void choosePartitionId()
      for (windowId entry in completeList)
         choosePartitionForEdges()
         remove entry from joinTable
         remove entry from completeList
```

Partition Edges

The final step of WinBro in Algorithm 1 sends all edges to the previously selected partitions with a *partitionCustom* function. With the given partition ID, Flink assigns the edge accordingly. Technically, this is no new *DataStream* but for demonstration, this last step is separated from the others. Consequently, the result is a data stream which can be further processed with any (streaming) application in Flink which uses edges in this format.

4.2 Hash Partitioner

By default, Flink partitions data with hashing (MurMurHash), resulting in good balance and speed also having high vertex cuts. In contrast to the parallel DAG with broadcasting state across machines, the implementation of this partitioner is simple, as presented in the straight line Flink Job graph in Figure 4.1. Below that, the operators are briefly described.

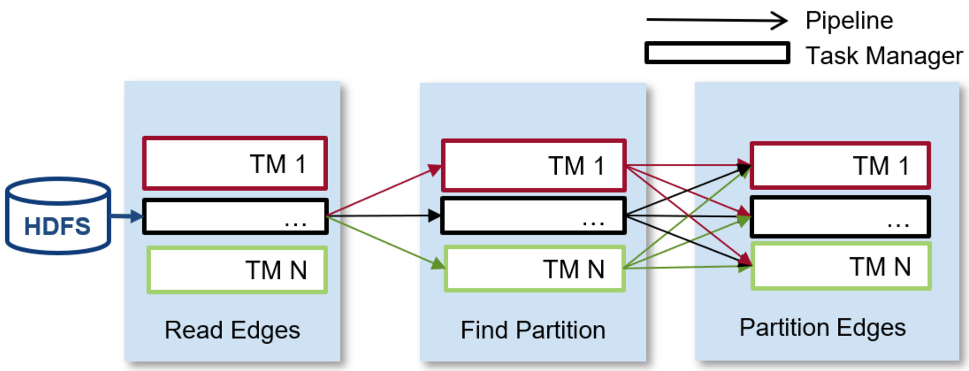


Figure 4.1: Parallel Hash Partitioner DAG

1. **Read Edges:** The read operator is similar to the operator used for the WinBro flow, i.e. it reads from the default file system or HDFS with the same parallelism limitations. The only difference is that there is no need to generate a windows ID because this DAG works without windows.

2. **Find Partition:** All edges get their partition ID in this operator. The partition ID is the modulo of the hashed source vertex value and the number of partitions, as presented in the Background section.

3. **Partition Edges:** In the last job vertex, the edge is sent to the assigned task manager which writes it to a sink.

The simplified DAG Java code to partition edges with a default murmurhash function looks as follows:

Algorithm 5 Hash Partitioning Flink DAG

```
// similar to read in WinBro DAG
SimpleEdgeStream<Edge> edgeInput = readTextFile(input);

DataStream<Edge> partitionerStream = edgeInput
      .partitionCustom(murmurHash(source.hashCode() %
         numOfPartitions));

DataStream<Edge> partitionedEdges =
   partitionerStream.addToSink();
```

4.3 State and Memory Optimization

A growing number of vertices and window IDs also increases the need for memory in the JVM. Thus, limiting the overall state size can either be achieved with a lower number of elements (objects). During the implementation phase three areas showed potential to save limit the memory consumption.

1. Number of entries in the join table
2. Dynamically state entry removal
3. State limitation with Reservoir Sampling

Join Table Cleanup

As written in Section 3.2.4, the window ID join mechanism causes for two different reasons: First, it requires the creation of edge IDs and window IDs in operators prior to the Match Function. Second, a high number of window ID join entries created by broadcast degree information is never updated. The reason is simple: The amount of broadcasts arriving at one task manager is proportional to the overall parallelism. This can be seen in the same Figure 3.6, on Task Manager 1 (TM1): Although all entries with incoming edges are marked complete, those with only broadcasts are not. With parallelism 2, around 50 percent of all entries are not updated, with parallelism 4 75 percent, and so on. In order to reduce this, a scheduled cleanup method is part of WinBro. This helps to reduce overhead by removing not updated map entries. Although the cleanup interval can be freely chosen, different small tests show that the waiting time between broadcasts and edges, and vice versa, is in milliseconds to seconds range. In order to be on the safe side, all experiments, the cleanup interval was set to several minutes.

Dynamic State Reduction

Another method of cleaning state is to dynamically remove vertex degree information during the partitioning process. The idea is to continuously measure the throughput of processed edges. Once it significantly slows down over a period of time, low degree vertices are removed with one-time operations. The rationale behind removing these vertices is the same as explained for Reservoir Sampling. However, this method is not yet fully developed and tested. With the current version of WinBro, this operation is triggered when a given threshold is reached. In this case, low degree vertices with a degree lower than 10 percent of the average degree are removed. Though, this method needs further analysis.

Limit State with Reservoir Sampling

Regular cleanup methods help to lower the memory consumption during the partitioning process but do not solve the general scaling issue of HDRF and DBH: the state size is at least as big as the number of vertices and their degrees. Thus, a possibility to reduce memory demands with increasing parallelism is to limit the state size with a representative sample of the stream. Reservoir Sampling [41] algorithm collects a representative sample of a data stream and statistically guarantees that all elements have the same probability to be added to the sample.
Reservoir Sampling works as follows: Until the sample size is full, every stream element is added to the sample. Once it is full, every newly arriving element is kept with a probability of (s/n) where s denotes the sample size and n is the number the arrived element. If this element is added to the sample, it replaces a randomly chosen sample element.
This method was implemented into WinBro and first tests were done but the current version of WinBro needs tuning with respect to removing high-degree nodes. Since high-degree nodes are important for DBH and HDRF, they should not be replaced as easily as low-degree vertices. Consequently, a centrality-based second check of sample replacement should be performed.

Chapter 5

Experiments

After having elaborated background and implementation for WinBro, this chapter presents the experimental setup, the test data sets, explains which experiments are executed and finishes with an experiment plan for a final overview. All results and their interpretations are shown in the next chapter.

5.1 Environment Setup

All experiments are conducted on a fully-distributed Flink cluster on four CentOS 7.6 servers. Every server has an Intel Xeon X5660 processor with 2.80 Gigahertz, and a total of 24 central processing units (CPU). Furthermore, every machine has 40 GB random access memory (RAM) and 6 TB of disk space. All machines are physically located in the same rack in a research data center and are interconnected via GigaBit network cables.
Apache Flink runs with version 1.7.1 with Java version 1.8. The cluster has one Job Manager and four Task Managers. As distributed file system acts HDFS (Hadoop 3.1.2) in a fully distributed way on the same servers.

5.2 Graph Data Sets

All experiments are executed on real-world social network graphs, with the exception of Skitter which is a power-law graph of autonomous systems connected to each other. The motivation behind this decision is that, as shown in Section 2.1, most real graphs have a power-law distribution. Also, HDRF and DBH perform well on power-law distribution.
These graphs are provided by the KONECT project from Koblenz University [14] that provides large data sets under *Creative Commons* license. Table 5.1

lists all graphs used for experiments. *TwitterSmall* is a subgraph of the Twitter data set, cut after 1.1 billion edges.

Table 5.1: Graph Data Sets for Experiments

Graph Name	**Vertices** (n)	**Edges** (m)
Skitter	1,696,415	11,095,298
Orkut	3,072,441	117,184,899
Twitter	41,652,230	1,468,365,182
TwitterSmall	36,901,926	1,100,000,000
Friendster	68,349,466	2,586,147,869

5.3 Metrics

Partitioning Quality

When recapping the goals of graph partitioning, the number of cuts should be low and all partitions should be equally balanced. Applying these two objectives to edge partitioners, two metrics are common:

The first indicator is **Replication Factor**, introduced in 2.5.2 which measures the total number of vertex copies compared to the total number of vertices in a graph. Generally, a low value is desirable for good partitioning results, indicating that the not many vertices are cut during the partitioning process.

The second metric in this category is the **load balance** of edges in all partitions, also shown in Section 2.5.2. For a perfect balance, the load balance should be exactly 1, indicating that all partitions received the same number of edges.

Partitioning Speed

The run time of a partitioning process is important to measure because it can influence the decision for or against an algorithm. To get a holistic view of the overall performance, the throughput is used as a metric, in edges per second. A second metric is the throughput per task manager. This allows measuring how the partitioner performs with an increasing parallelism. For both criteria, the objective is to have a high throughput of edges, showing a low total partitioning time. Hence, the following two metrics are used for the speed:

- Throughput of edges per second
- Throughput of edges per second per task manager

Streaming Applications

Although measuring the quality and speed of partitioning algorithms can provide good insights for different application areas and objectives, one should bear in mind that running a partitioning algorithm is usually an intermediate step for distributed applications. Consequently, the actual goal is to use partitioned (sub)graph data as input for concrete computations. In order to see the performance of applications using WinBro, two different programs, namely Connected Components and Bipartiteness Check, are run with subgraphs produced by WinBro as input. This is compared with Hash-based partitioned data input.
The experiments look as follows: A streaming application with an integrated version of WinBro is run on different graph data sets and different degree of parallelism in Apache Flink. The metric to measure is the throughput of edges per second, recorded after different run times. The goal in this streaming setup is not to finish the partitioning process because it shall simulate an unbounded stream.
The following two algorithms are used to validate WinBro with streaming applications

1. **Connected Components:** The connected components application uses an edge stream input to identify all components of a graph. For example, when all vertices can be reached from all other vertices, one large component exists. However, graphs can be disjoint where communities are not connected to each other. The algorithm works as follows: It keeps adding edges to disjoint data sets and checks both vertices for neighbors in the assigned component. If these vertices co-exist in both sets, they are merged into one because this edge connects them. When running this algorithm in a distributed manner, regular merge operations attempt to find cross-partition components to get a global picture of all connected components. The assumption is that partitions with a low replication factor result in a smaller number of connected components per instance because their locality is already reflected during the partitioning process. This decreases the number of needed merge operations.

2. **Bipartiteness Check:** A graph is bipartite if it has two vertices types or groups, where one vertex is linked to other vertices it belongs to. Usually, these vertices are graphically represented as two opposing groups of nodes, as shown in Figure 5.1. What makes this graph special is that no connection exists within one group. A common example of a bipartite network is an actor and movie graph where actors point to movies they play a role in [10]. With regard to this application, all vertices of one group must point to the other vertex group. If edges exist within one group, a graph is not bipartite. The Bipartite Check algorithm adds edge vertices to these groups, always checking whether this new vertex keeps the bipartite structure. If not, it is marked as non-bipartite. The parallel version of this algorithm works in a similar way but adds periodical updates across partitions and merges these groups accordingly. This ensures that the global graph is examined for bipartiteness.

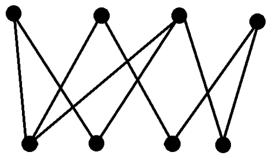


Figure 5.1: Bipartite graph

5.4 Overview of all experiments

After having described different metrics, this closing section provides an overview of all experiments conducted for WinBro. In general, every setting compares the following three algorithms implemented in WinBro: HDRF, DBH and Hash. Furthermore, all experiments were run at least three times to verify the results where the average value is taken for the result.

Replication Factor and Load Balance

The experiment plan to measure Replication Factor and Load Balance is presented in Table 5.2. Due to limited computational power and memory (4 servers with 40 GB RAM each), the big graphs (Twitter and Friendster) are not partitioned with parallelism higher than 4. This is discussed in the Results section.

Table 5.2: Replication Factor Experiment Plan

	Parallelism (k)				
Graph	**2**	**4**	**8**	**16**	**24**
Skitter	x	x	x	x	x
Orkut	x	x	x	x	-
Twitter(small)	x	x	-	-	-
Twitter	x	x	-	-	-
Friendster	x	x	-	-	-

Partitioning Speed

The experiment plan to measure the partitioning speed is similar to the one measuring load and cut, and looks as in Table 5.3:

Table 5.3: Partitioning Speed Experiment Plan

	Parallelism (k)				
Graph	**2**	**4**	**8**	**16**	**24**
Skitter	x	x	x	x	x
Orkut	x	x	x	x	-
Twitter(small)	x	x	-	-	-
Twitter	x	x	-	-	-
Friendster	x	x	-	-	-

Streaming Application Performance

The setup to test WinBro with streaming applications is presented below. In contrast to the two other categories mentioned above, streaming a big graph is feasible here because the applications never complete since they are made for streams. Instead, the throughput is recorded at different checkpoints, shown in Table 5.4.

Table 5.4: Streaming Applications Experiment Plan

	Minutes			
Graph	**15**	**30**	**45**	**60**
Twitter	x	x	x	x

Chapter 6

Results & Discussion

This chapter shows the experimental results for comparing WinBro using HDRF and DBH with results from Hash partitioning in terms of partitioning quality, speed and the effect of these partitioning methods for various streaming applications. Unless differently marked, the structure is similar to the Experiment chapter. Every category is presented separately with all experiments, followed by a brief summary, and closing with a discussion.

6.1 Partitioning Quality

Replication Factor

Twitter

The replication factor of Twitter with a parallelism of 4 is shown in Figure 6.1. DBH shows the lowest cuts, followed by HDRF, whereas Hash cuts the highest number vertices. WinBro was unable to partition Friendster with HDRF, see more details below the graph.

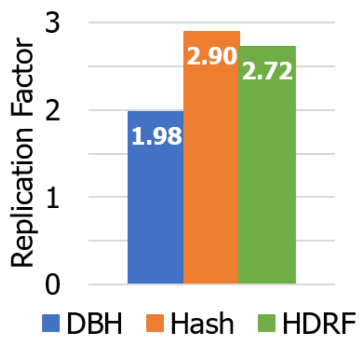


Figure 6.1: Replication Factor of Twitter Graph (parallelism 4)

The significance of the results is low because all experiments were only run once or twice, partly giving different results. Thus, a general statement needs more experiments. Furthermore, due to inference on the experiment cluster and limited memory capacity, WinBro was unable to partition with parallelism higher than 4, even with Hash. Moreover, the replication factor of Friendster could not be calculated due to the same memory constraints even though the partitioning process was successful for DBH and Hash. Furthermore, HDRF and DBH failed multiple times with parallelism of 2 for both Twitter and Friendster. Thus, these results are omitted.

TwitterSmall

Figure 6.2 shows the replication factor of HDRF, Hash, and DBH for the TwitterSmall graph with 1.1 billion edges for parallelism two and four. The best cuts are provided by DBH on both levels of parallelism with replication factors of 1.28 and 1.71 respectively. The replication factor of HDRF ranks second with 1.52 and 2.46. The highest number of cuts is done by Hash which replicates vertices with a factor of around 1.74 and 2.92.
Overall, the results show that replication is higher when parallelism is increased. Also here, the environmental setup did not allow a higher parallelism to finish the experiments due to too high memory load.

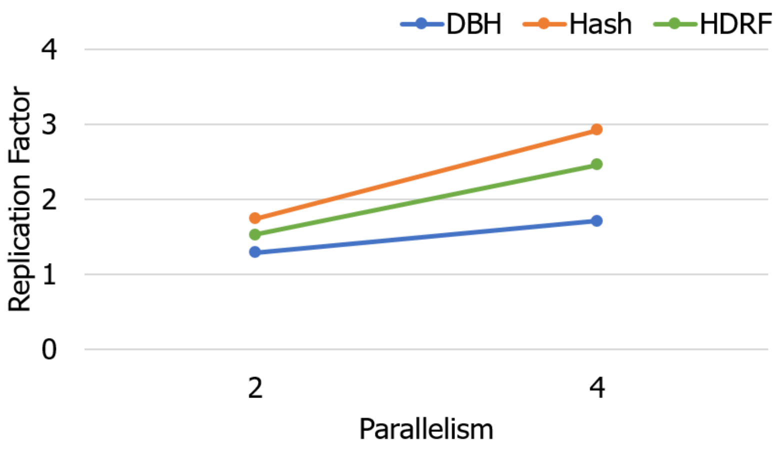


Figure 6.2: Replication Factor on TwitterSmall Graph

Orkut
When studying the replication factor for different levels of parallelism for Orkut in Figure 6.3, it shows that DBH and HDRF always perform better than Hash and the replication factor increases with a higher level of parallelism for all three algorithms, although parallelism gives nearly identical results. It is visible that the curve of DBH is not as steep as the one of HDRF or Hash. Thus, DBH's vertex cuts increase slower compared to HDRF and Hash.

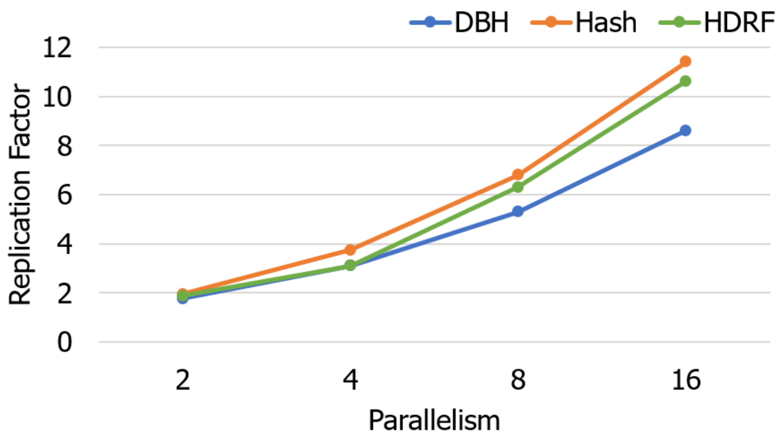


Figure 6.3: Replication Factor on Orkut Graph

Skitter
The results for Skitter graph are presented in a diagram in Figure 6.4. Skitter

is the smallest of all test graphs and it was possible to go up to parallelism 24 with the cluster for the experiments. The Hash partitioner could even succeed to partition with parallelism 32. For every increasing level of parallelism, the number of cuts for all algorithms increases. Though, the slope of DBH is not as steep as the one of HDRF and Hash. Throughout all levels of parallelism DBH performs best, followed by HDRF and Hash.

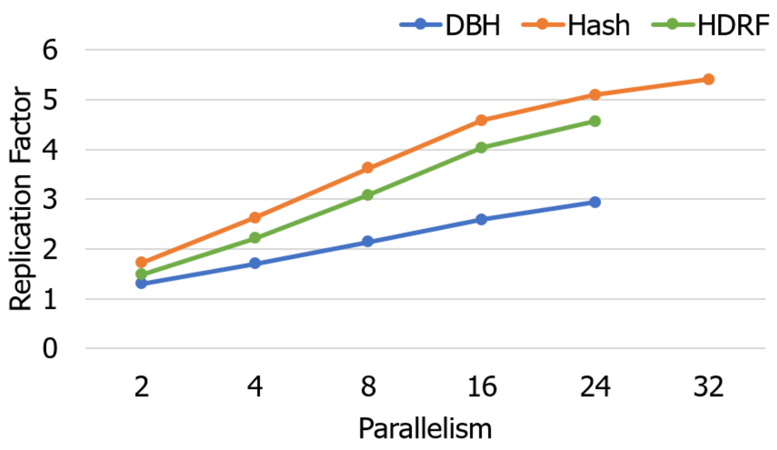


Figure 6.4: Replication Factor on Skitter Graph

Load Balance

Twitter
Due to the same issues faced with the replication factor, Friendster and Twitter are only partitioned on low parallelism, also missing HDRF for Friendster. Similarly, the load could not be calculated for Friendster. Regarding Twitter in Table 6.1, it is visible that the load is nearly perfect for all algorithms on both graphs because it is very close to 1.0.

Table 6.1: Twitter Load Balance (4 partitions)

Algorithm	**Load Balance**
DBH	1.01
Hash	1.01
HDRF	1.00

TwitterSmall

For the TwitterSmall graph, all algorithms produce practically the same load balance on both parallelism 2 and 4. Hence, hardly any difference is remarkable in Table 6.2:

Table 6.2: TwitterSmall Load Balance

	Partitions	
Algorithm	**2**	**4**
DBH	1.02	1.00
Hash	1.00	1.01
HDRF	1.00	1.00

Orkut

Orkut graph data, similar to TwitterSmall, does not indicate differences concerning the load balance, as presented in Table 6.3, even though four different levels of parallelism up to 16 were tested.

Table 6.3: Orkut Load Balance

	Partitions			
Algorithm	**2**	**4**	**8**	**16**
DBH	1.00	1.00	1.00	1.00
Hash	1.00	1.00	1.01	1.01
HDRF	1.00	1.00	1.00	1.00

Skitter

The final partitions of the Skitter graph are evenly balanced for all levels of parallelism for DBH and HDRF. A different picture can be seen for Hash, as visible in Table 6.4. While it partitions almost perfectly balanced on parallelism 2 and 4, the load imbalance increases especially with parallelism 16 and 24, where it reaches 1.08 and 1.20 respectively.

Table 6.4: Skitter Load Balance

	Partitions				
Algorithm	**2**	**4**	**8**	**16**	**24**
DBH	1.00	1.00	1.00	1.00	1.01
Hash	1.00	1.02	1.04	1.08	1.20
HDRF	1.00	1.00	1.00	1.00	1.03

In the following, the main findings of this category are summarized. Below that, the results are discussed:

- The main observation is that increasing parallelism also increases the number of cuts for all three algorithms.

- For all levels of parallelism, DBH provides the best cuts on all graphs. HDRF gives the second-best cuts whereas Hash creates the highest number of vertex copies.

- The load balance of all three algorithms is very close to 1.0 in most experiments, but for Skitter an increased load imbalance is observed for Hash algorithm with parallelism 16 and 24.

The results of this category can be explained as follows: With a higher number of partitions and increasing parallelism, edges must be distributed to more machines. Hence, the graphs are divided into more subgraphs, which causes more cuts and a higher replication factor. This finding is confirmed in different studies [3] [4].
However, DBH performs better than HDRF is most of the cases. This differs from other studies. The reason is that partitioners in these studies were either non-parallel or used shared state mechanisms for DBH and HDRF. WinBro runs in parallel without a shared state instead. As a result, HDRF has no synchronization mechanism after the degree count. Thus, two important factors contributing to the HDRF score, machine load and partition assignment, are not reconciled in a later stage. This leads to different results and thus higher cuts. DBH performs better because it sends edges to partitions based on the hash value of the vertex with the lower degree, a piece of information which is fully available to the global state in every parallel instance. To a certain extent, this is more consistent The result of the Hash partitioner with the highest cuts can be explained with the model-agnostic implementation of Hash, as explained earlier in the thesis. Furthermore, this finding is confirmed by other

studies. Finally, the load balance is overall very good with nearly all levels of parallelism and graphs showing an almost perfect load balance of 1.0. Only for Skitter, Hash provides a worse load for a high level of parallelism. One reason can be that the graph is skewed, i.e. not densely connected. With simple hashing, this can lead to an uneven load balance.

6.2 Partitioning Speed

Throughput in Edges per Second

All Graphs

Figure 6.5 shows the duration of a complete partitioning process for different graphs, not considering different levels of parallelism. The run time is displayed in throughput (edges per second). Thus, a higher throughput refers to a lower partitioning time.

Hash shows the highest throughput of edges for all graphs. DBH has the second highest speed on all graphs but Skitter, where it is 1000 edges per second slower than HDRF. Besides this, HDRF always shows the lowest throughput but is partly very close to DBH.

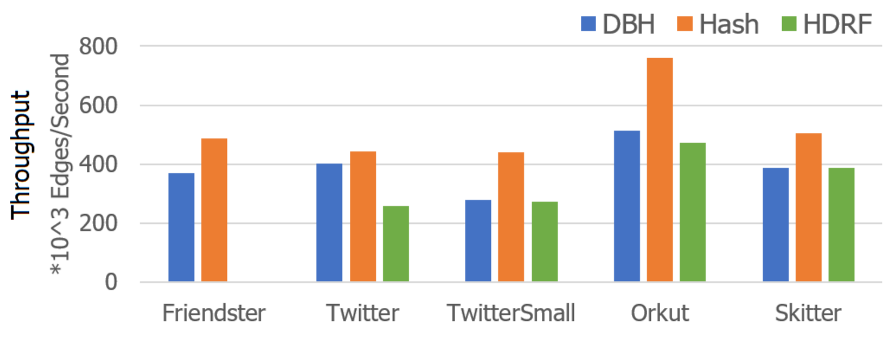


Figure 6.5: Edge Throughput per Algorithm for all Graphs

Since Twitter and Friendster were only successfully tested with a parallelism of 4, a separate graph is omitted. Moreover, even though some of these results are very clear, it must be admitted that the variation of execution times differs significantly for the same experiments. This is caused by interference with other processes simultaneously running on the test servers.

Throughput in Edges per Second per Task Manager

TwitterSmall

TwitterSmall could only be partitioned on with parallelism 2 and 4, as visible in Figure 6.6 but it is sufficient to provide results here. The number of edges per second increases for Hash from 125,000 to 160,000 with increasing parallelism whereas DBH achieves nearly the same throughput for parallelism 2, but drops instead to roughly 80,000 edges per second with increasing parallelism. HDRF has the highest execution time, also with declining throughput from 90,000 and 80,000 edges.

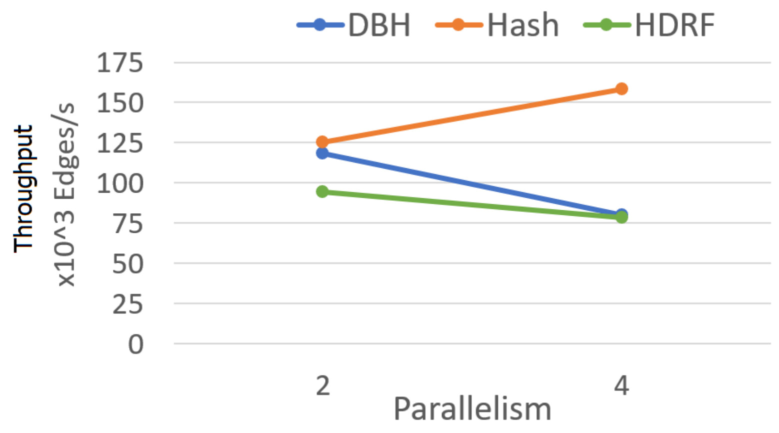


Figure 6.6: Throughput on TwitterSmall per Task Manager by Parallelism

Orkut

The throughput of edges per task manager for parallelism 2 to 16 for the Orkut graph can be seen in Figure 6.7. With parallelism 2, all algorithms show a comparable processing speed of 100,000 to 120,000 edges per second. However, when parallelism increases, the performance of HDRF and DBH continuously drops, until to roughly 40,000 edges per second for HDRF and 50,000 for DBH. A different observation is made for Hash. It gives almost the same throughput, behaving nearly agnostic to parallelism. Throughout all levels of parallelism, the edges per second range between 96,000 and 110,000.

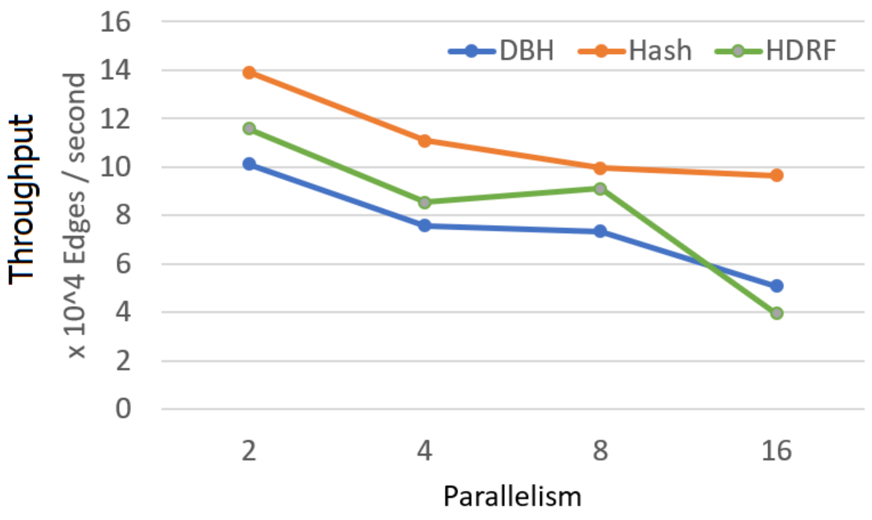


Figure 6.7: Throughput on Orkut per Task Manager by Parallelism

Skitter

The same metric looks different for the Skitter graph, see Figure 6.8. In this case, the throughput of all algorithms decreases with increasing parallelism. It is visible that their trendlines form the same shape. Hash is always partitioned faster than DBH and HDRF though. DBH and HDRF produce nearly the same throughput but usually, process around 15,000 to 20,000 edges less per second. Only with parallelism 24, DBH and HDRF approach Hash.

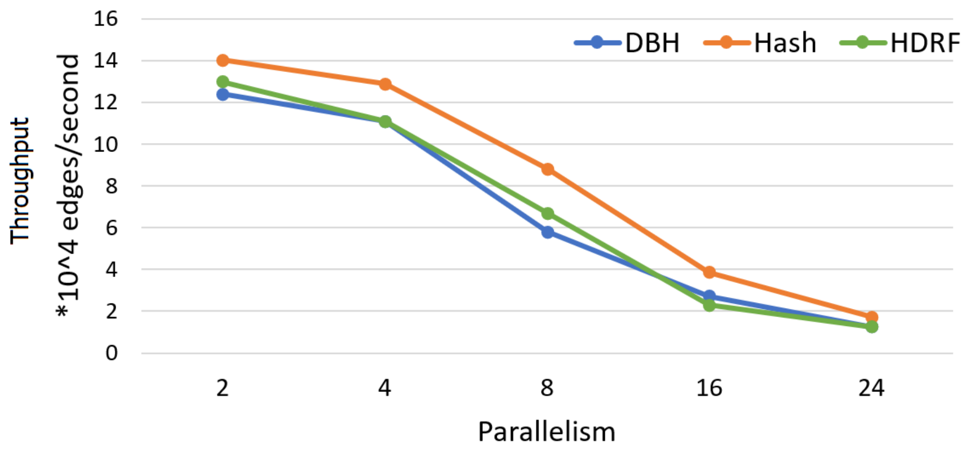


Figure 6.8: Throughput on Skitter per Task Manager by Parallelism

Before starting the discussion of the processing speed, the speed results can be summarized as follows:

- In all cases, Hash partitioning is faster than DBH and HDRF, especially when parallelism increases.
- The results for DBH and HDRF are different; they share the second and third rank depending on the graph.

The obvious result of a high-performing Hash algorithm can be justified with its lightweight implementation. The decision for an edge partition assignment is state-agnostic so that no information such as vertex degree or previously assigned edges is required. This makes it different from DBH which also uses hash but requires degree information of the vertices. The consequence is that it must be collected and also searched. HDRF does not only collect degree information but also all vertex assignments and the partition load.
The increased lookup and calculation time is not the only reason for the poorer performance of HDRF and DBH. Another reason why they perform worse with increasing parallelism, is the increasing demand for memory when the state grows. As mentioned in the test setup, the experiments were run on four 40 GB RAM machines. Once the parallelism was set to 8 and higher, this memory had to be shared between at least two task managers. With the maximum parallelism of 24, 6 task managers were sharing one physical server.
Furthermore, especially big graphs partitioned with DBH and HDRF reached the machines' memory limits quickly. Consequently, the underlying operating system triggered swapping mechanisms which slowed down the process even more, resulting in increasing run time.

6.3 Streaming Applications

After having evaluated experiments with an isolated view on the partitioning performance and quality, the next step is to integrate WinBro into real streaming application to measure its capabilities to provide good input data for graph analysis tasks.
Connected Components
Figure 6.9 displays the number of edges processed per second during Connected Components application on Twitter. Snapshots are taken every 15 minutes. DBH has the highest throughput in the beginning with almost 30,000 edges per second, whereas HDRF processed nearly 25,000 edges/second until then and Hash has even 2000 less. The second snapshot after 30 minutes

shows that the throughput decreases in total to 15,000 to 20,000 with the same order. At the next two checkpoints at 45 and 60 minutes, the ranking does not change but HDRF approaches DBH and Hash falls behind, so that only 36 Million edges were processed in total for Hash. In contrast, DBH and HDRF processed 55 and 50 million edges at the same time.

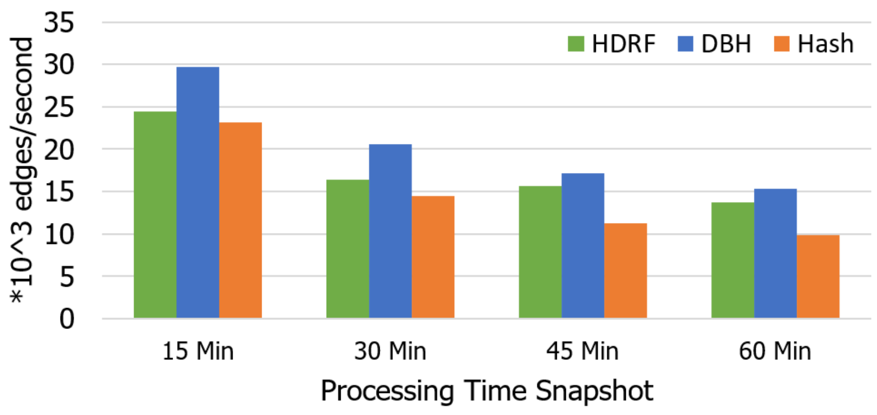


Figure 6.9: Connected Components on Twitter Graph, parallelism 4

Bitpariteness Check

The second streaming application for WinBro evaluation is Bipartiteness Check. Figure 6.10 provides an overview of the results for Twitter parallelism 4. In this case, only a single snapshot after 15 minutes is necessary to identify the best performing algorithm: Hash processed an average of 1.6 Million edges per second, nearly double the speed of HDRF and DBH which processed in average 700,000 and 850,000 edges per second in the same time respectively. After 17 minutes, the whole Twitter graph was partitioned with Hash.

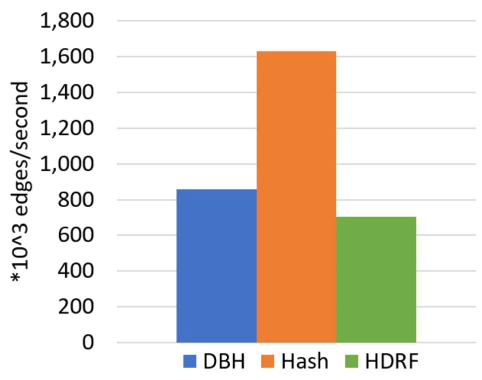


Figure 6.10: Bipartiteness Check on Twitter Graph, parallelism 4

As a summary of the streaming applications, the following is observed:

- DBH gives the best performance for Connected Components, followed by HDRF. Hash is behind these two algorithms.

- When running a Bipartiteness Check with WinBro, Hash outperforms DBH and HDRF by factor 2. The latter two have a comparable result, although DBH is slightly faster.

The good result for DBH and HDRF seen with the Connected Components algorithm comes from the need for data-locality to identify neighboring vertices and to receptively assign a common value to a local community. This leads to high communication cost if many vertices are replicated. Both DBH and HDRF give low cuts, as shown previously in this chapter. Consequently, the Connected Components application benefits from this effect. On the other side, Hash does not consider locality when choosing a partition, leading to higher communication cost between vertices.
Concerning Bipartiteness Checker, the outstanding result of the Hash partitioner can be explained with the requirements for the Bipartiteness Check algorithm. In contrast to Connected Components, the merge phase requires only a small amount of data exchange in every merge phase, practically only both vertex groups in form of a list with the label whether it is bipartite or not. The consequence is a faster merging phase and less communication than required by Connected Components.

6.4 Summary

Different sets of experiments with WinBro either HDRF or DBH algorithm were made, always in comparison with a regular Hash partitioner. These experiments used real graphs, all of them following a power-law degree distribution. The metrics measured included partitioning quality, speed, and performance in different streaming applications. Because WinBro is a parallel partitioner, most of these experiments were run for different levels of parallelism.

Partitioning Quality

With regard to the partitioning quality and **replication factor**, results provide evidence that Hash produces more vertex-cuts than HDRF and DBH, and thus, has a lower quality of partitioning. This is the case for all networks and all levels of parallelism. DBH provides the best partitioning quality because it shows the lowest cuts overall. HDRF can be found between DBH and Hash. This can be justified missing synchronization of edge assignments and machine load after the initial partitioning decision in the parallel HDRF version. Hence, unaligned decisions on parallel running instances are possible. This drawback is not observed with DBH which works degree-based but eventually uses a hashing mechanism for the edge assignment. Thus, it provides more consistency with regard to final partition assignments. Nevertheless, the replication factor always increases when the level of parallelism increases. This observation is covered by different studies.
The second metric, **load balance**, is very good with 1.0 for all graphs and nearly all levels of parallelism so that good quality overall is concluded. However, more experiments with higher parallelism, especially on large graphs could help to validate the findings in this category.

Partitioning Speed

All experiments to measure the throughput of edges during the partitioning process give a clear answer, especially when looking at higher levels of parallelism. The Hash partitioner outperforms HDRF and DBH. The reason is that it does not require state information and can run fast even with a high system load. An increasing number of edges does not influence the speed, thus the edges processed per second are nearly constant. The opposite is the case for HDRF and DBH. Since both need historical knowledge about the graph, more arriving edges increase the cost of a partition assignment. This leads to the issue that their state requirements are proportional to the graph size. With regard to parallelism, two options exist to have the same performance for increasing parallelism: Either resources (mainly memory) is added proportionally, or the state size must be limited to constant size without a loss of partitioning quality.

Streaming Applications

The final category of experiments was the integration of WinBro into streaming applications to see if their presumably better cuts help to improve the performance of the applications: This was the case for the Connected Components application which benefits from good cuts when running in parallel because the result is a smaller number of partial components. Thus, fewer merging operations need to be performed on the inter-cluster level. Consequently, DBH and HDRF always increase the throughput of edges compared to Hash, showing that higher partitioning time is compensated with faster processing time in the application afterward.

When integrating WinBro before Bipartiteness Check application, Hash performs much better than DBH and HDRF, i.e. it provides a higher throughput of edges. This can be justified with the actual merge phase of a Bipartiteness Check. As explained earlier, every phase only exchanges two lists of vertex lists without complex requirements. This does not significantly interrupt the process and Hash can continue faster.

Table 6.5 provides an overview of the different categories with the best performing algorithms, based on the findings from the experiments:

Table 6.5: Result Table for Parallel Algorithms with WinBro

Category	**Preferable Algorithm**
Cuts	DBH
Speed	Hash
Load	all
Apps relying on locality	HDRF/DBH
Apps agnostic to locality	Hash

Chapter 7

Conclusion

The main objective of this thesis was to implement a parallel graph partitioner for a stream processing system without a shared state mechanism based on well-performing state-based partitioning algorithms HDRF and DBH. The second objective was to find out whether this parallel partitioner produces better partitioning results on real-world graphs compared to the default Hash-based partitioner.
After having evaluated the experiments, the main objective can be concluded as fulfilled. This work's contribution is WinBro, a DBH and HDRF based parallel edge partitioner developed in Java and fully integrated into Apache Flink, a pure stream processing system. WinBro uses broadcasts to share degree information across the streaming pipeline. Furthermore, HDRF and DBH algorithm were implemented in the way that they can run in parallel.
To validate the second objective regarding the comparison of WinBro with Hash partitioning, experiments were conducted with different levels of parallelism and on different real-world graph data sets. The following results are important:
As expected, Hash-based partitioning has the highest speed, also for streaming applications without the need for well-partitioned input streams. However, it gives a low partitioning quality. In contrast, WinBro, either with DBH or HDRF, results in lower vertex cuts. The better quality pays off in streaming applications relying on intact structures, where WinBro has an overall shorter run time than Hash.
These findings lead to the following conclusion: When using WinBro for real-world graphs following power-law degree distribution, it gives better results than the default Hash partitioner in Apache Flink. Even more important, WinBro improves the performance of streaming graph applications requiring well-

partitioned input data.
Although this conclusion proves WinBro a good performance, the scalability of WinBro is bound to state requirements of HDRF and DBH and needs to be addressed in the future.

7.1 Future Work

Although WinBro is generally verified, different limitations could be identified and need to be addressed in the future. It is possible to categorize them as state, scalability, and algorithm. They are described in the following paragraphs:

State: Addressing the state size of HDRF and DBH is crucial for parallel execution because the current implementation replicates the whole state on every task manager. Different possibilities to reduce this state need further examination. A first step was the implementation of Reservoir Sampling but it requires a better tuning. It might be necessary to change certain parameters or to find other ways to reduce the state without lowering the replication factor or increasing the partitioning time. Different ideas for state reduction are possible but need exploration, e.g. caching high-degree vertices combined with shared-state, or adding another vertex eviction policy with event-based cleanup methods. A very recent publication from Hua et al. [42] suggests performing parallel streaming graph partitioning with a game theory approach. This approach promises very little state requirements and could be taken into account in the future.

Scalability: The main challenge for the experiments was dealing with the scalability. In the future, bigger graphs need to be tested with more powerful machines to continue the development and to increase the significance of the results.

Algorithm: The actual procedure of WinBro does exchange further state information after the degree broadcast. It might be beneficial for the partitioning quality to synchronize machine loads or assigned partitions. One potential solution for more data exchange is Flink's experimental new feature of Iterative Streaming. Also, initial edge keying/hashing after reading could be improved by sending edges to predictable nodes, so that every task manager is responsible for a certain range of edges. However, both options must be thoroughly considered and tested.

Bibliography

[1] Jack Loechner. *90% Of Today's Data Created In Two Years*. URL: https://www.mediapost.com/publications/article/291358/90-of-todays-data-created-in-two-years.html (visited on 07/20/2019).

[2] *Facebook Tops 1 Billion Monthly Active Users, CEO Mark Zuckerberg Shares A Personal Note*. TechCrunch. URL: http://social.techcrunch.com/2012/10/04/facebook-tops-1-billion-monthly-users-ceo-mark-zuckerberg-shares-a-personal-note/ (visited on 07/22/2019).

[3] Zainab Abbas et al. "Streaming graph partitioning: an experimental study". In: vol. 11. 11. July 1, 2018, pp. 1590–1603. DOI: 10.14778/3236187.3236208. URL: http://dl.acm.org/citation.cfm?doid=3236187.3269471 (visited on 01/17/2019).

[4] Isabelle Stanton and Gabriel Kliot. "Streaming graph partitioning for large distributed graphs". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*. the 18th ACM SIGKDD international conference. Beijing, China: ACM Press, 2012, p. 1222. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339722. URL: http://dl.acm.org/citation.cfm?doid=2339530.2339722 (visited on 05/26/2019).

[5] Fabio Petroni et al. "HDRF: Stream-Based Partitioning for Power-Law Graphs". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management - CIKM '15*. the 24th ACM International. Melbourne, Australia: ACM Press, 2015, pp. 243–252. ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806424. URL: http://dl.acm.org/citation.cfm?doid=2806416.2806424 (visited on 05/26/2019).

[6] *Data protection*. European Commission - European Commission. URL: https://ec.europa.eu/info/law/law-topic/data-protection_en (visited on 07/22/2019).

[7] Hiroshi Yamaguchi et al. “Privacy preserving data processing”. In: *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE. 2015, pp. 714–719.

[8] Nicola Jones. “How to stop data centres from gobbling up the world’s electricity”. In: *Nature* 561 (Sept. 12, 2018), p. 163. DOI: 10.1038/d41586-018-06610-y. URL: http://www.nature.com/articles/d41586-018-06610-y (visited on 07/22/2019).

[9] Chris Godsil and Gordon F Royle. *Algebraic graph theory*. Vol. 207. Springer Science & Business Media, 2013.

[10] Mark Newman. *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc., 2010. ISBN: 0199206651.

[11] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*. Vol. 290. Citeseer, 1976.

[12] Joseph E Gonzalez, Yucheng Low, and Haijie Gu. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: (), p. 14.

[13] Erica Klarreich. *Scant Evidence of Power Laws Found in Real-World Networks*. Quanta Magazine. URL: https://www.quantamagazine.org/scant-evidence-of-power-laws-found-in-real-world-networks-20180215/ (visited on 03/07/2019).

[14] Jérôme Kunegis. “KONECT: The Koblenz Network Collection”. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13 Companion. Rio de Janeiro, Brazil: ACM, 2013, pp. 1343–1350. ISBN: 978-1-4503-2038-2. DOI: 10.1145/2487788.2488173. URL: http://doi.acm.org/10.1145/2487788.2488173.

[15] Hooman Peiro Sajjad et al. “Boosting Vertex-Cut Partitioning for Streaming Graphs”. In: *2016 IEEE International Congress on Big Data (BigData Congress)*. 2016 IEEE International Congress on Big Data (BigData Congress). San Francisco, CA, USA: IEEE, June 2016, pp. 1–8. ISBN: 978-1-5090-2622-7. DOI: 10.1109/BigDataCongress.2016.10. URL: http://ieeexplore.ieee.org/document/7584914/ (visited on 01/31/2019).

[16] Kook Jin Ahn and Sudipto Guha. “Graph Sparsification in the Semi-streaming Model”. In: *Automata, Languages and Programming*. Ed. by Susanne Albers et al. Vol. 5556. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 328–338. DOI: 10.1007/978-3-642-02930-1_27. URL: http://link.springer.com/10.1007/978-3-642-02930-1_27 (visited on 07/22/2019).

[17] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (Jan. 1, 2008), p. 107. ISSN: 00010782. DOI: 10.1145/1327452.1327492. URL: http://portal.acm.org/citation.cfm?doid=1327452.1327492 (visited on 07/20/2019).

[18] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 07/22/2019).

[19] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[20] Paris Carbone et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[21] *Benchmarking Streaming Computation Engines at Yahoo!* Yahoo Engineering. URL: https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at (visited on 07/20/2019).

[22] Aljoscha Krettek. *The Curious Case of the Broken Benchmark: Revisiting Apache Flink® vs. Databricks Runtime*. URL: https://www.ververica.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime (visited on 07/20/2019).

[23] *Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems - The Databricks Blog*. Databricks. Oct. 11, 2017. URL: https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html (visited on 07/20/2019).

[24] Matei Zaharia et al. “Discretized streams: Fault-tolerant streaming computation at scale”. In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM. 2013, pp. 423–438.

[25] *Apache Flink 1.8 Documentation: Apache Flink Documentation*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/ (visited on 07/22/2019).

[26] *Apache Flink 1.7 Documentation: Component Stack*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.7/internals/components.html (visited on 07/20/2019).

[27] *Apache Flink 1.8 Documentation: Dataflow Programming Model*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/programming-model.html (visited on 07/22/2019).

[28] Fabian Hueske and Vasiliki Kalavri. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. O'Reilly Media, 2019.

[29] Kurt Mehlhorn and A. Tsakalidis. "Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity". In: 1990.

[30] George Karypis and Vipin Kumar. "Multilevel Graph Partitioning Schemes." In: Jan. 1995, pp. 113–122.

[31] Charalampos Tsourakakis et al. "FENNEL: streaming graph partitioning for massive scale graphs". In: *Proceedings of the 7th ACM international conference on Web search and data mining - WSDM '14*. the 7th ACM international conference. New York, New York, USA: ACM Press, 2014, pp. 333–342. ISBN: 978-1-4503-2351-2. DOI: 10.1145/2556195.2556213. URL: http://dl.acm.org/citation.cfm?doid=2556195.2556213 (visited on 05/26/2019).

[32] Jaewon Yang and Jure Leskovec. "Defining and evaluating network communities based on ground-truth". In: *Knowledge and Information Systems* 42.1 (2015), pp. 181–213.

[33] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. "GraphBuilder: scalable graph ETL framework". In: *GRADES*. 2013.

[34] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. "X-stream: Edge-centric graph processing using streaming partitions". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 472–488.

[35] Julian Shun and Guy E Blelloch. “Ligra: a lightweight graph processing framework for shared memory”. In: *ACM Sigplan Notices*. Vol. 48. 8. ACM. 2013, pp. 135–146.

[36] Grzegorz Malewicz et al. “Pregel: a system for large-scale graph processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[37] Shiv Verma et al. “An experimental comparison of partitioning strategies in distributed graph processing”. In: *Proceedings of the VLDB Endowment* 10.5 (2017), pp. 493–504.

[38] *Apache Flink: Introducing Gelly: Graph Processing with Apache Flink*. URL: `https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html` (visited on 07/22/2019).

[39] *Apache Flink 1.8 Documentation: Apache Kafka Connector*. URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.8/dev/connectors/kafka.html` (visited on 07/22/2019).

[40] Apache Flink. *Apache Flink 1.7 Documentation: The Broadcast State Pattern*. URL: `https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/stream/state/broadcast_state.html` (visited on 07/11/2019).

[41] Jeffrey S. Vitter. “Random Sampling with a Reservoir”. In: *ACM Trans. Math. Softw.* 11.1 (Mar. 1985), pp. 37–57. ISSN: 0098-3500. DOI: `10.1145/3147.3165`. URL: `http://doi.acm.org/10.1145/3147.3165`.

[42] Q. Hua et al. “Quasi-Streaming Graph Partitioning: A Game Theoretical Approach”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.7 (July 2019), pp. 1643–1656. ISSN: 1045-9219. DOI: `10.1109/TPDS.2018.2890515`.

[43] *Apache Flink 1.8 Documentation: Process Function (Low-level Operations)*. URL: `https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/process_function.html` (visited on 07/12/2019).

[44] *Apache Flink 1.8 Documentation: Process Function (Low-level Operations)*. URL: `https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/process_function.html` (visited on 07/12/2019).

Appendix A

Appendix

A1 Join Operation Alternatives

For regular windows, Flink provides a window join function [43] but it is limited to two keyed streams. Thus, there is no out-of-the-box join operation available for broadcast streams in combination with windowed keyed streams. Broadcast streams are not keyed because their nature requires to send information to all nodes. Consequently, it is necessary to find a suitable joining method, and all options below were validated for effectiveness and efficiency:

1. Add an edge to a queue if vertex degree information is unavailable
2. Low level joins with Flink *ValueState* in CoProcess function
3. Join with the aid of watermarks
4. Join with the aid of a custom Window ID

The first approach was implemented in the beginning and was effective for small graphs. When a new broadcast element arrived, the whole list was iterated over and edges whose vertex information arrived were partitioned accordingly. However, this approach has two main disadvantages. First, every iteration has a complexity of *o(w)* where w is the number of waiting edges. With an increasing number of unprocessed edges, this leads to significant delays. Second, this method has no matching logic but iterates without any indication that (many) edges can be processed after having added one additional broadcast to the state. The window ID solution has an advantage compared to this waiting edges approach. It only iterates over edges once their equivalent broadcasts arrived. This ensures a single call per edge before getting assigned

to a partition. With *waiting edges*, it was repeated once per arriving broadcast element, i.e. at least once, but much more often in reality.
The official Flink documentation suggests implementing low level operations on a **CoProcess** function in combination with timer [44]. Attempts to implement this solution did not lead to success, either. The reason is that timers can only be used by keyed streams (not to broadcast). Different ways to work around this limitation did not solve the problem. One result was that a significant amount of edges was never partitioned but never left this operator. Second, when a timer call did not lead to emitting edges, it had to be repeated. The consequence was a very high number of unnecessary timer calls, that eventually slowed down the partitioning process.
Joins with **watermarks** are also no suitable option. This lies in Flink's implementation of watermarks. Although all edges have the same watermark when they leave the Read Edges operator, every downstream operator creates a new watermark stamp on the edge metadata. Consequently, there is no guarantee that broadcasts and their counterpart edges have the same watermark when arriving from the Read and Re-Label Edges operator. This behavior could quickly be observed during the first tests, confirming that it is no reliable joining option.

A2 Parallel DBH

All changes compared to the original version are marked with commands (green). The code is written in Java but slightly simplified to improve readability.

```
// DBH Select Partition
public int selectPartition(Edge edge) \{
      if (state.contains(source)
         first_vertex = state.get(source)
      else
         first_vertex = new DummyEdge // dummy added
     if (state.contains(target)
         second_vertex = state.get(target)
      else
         second_vertex = new DummyEdge // dummy added
     degree_u = first_vertex.getDegree(); // removed +1
        here
     degree_v = second_vertex.getDegree(); // removed
        +1 here
```

```
    if (degree_u < degree_v)
      machine_id = hash(firstVertex) \%
          numOfPartitions;
    else if (degree_v < degree_u)
      machine_id = hash(second_vertex) \%
          numOfPartitions;
    else
      machine_id = random(numOfPartitions)
  return machine_id;
```

A3 Parallel HDRF

All changes compared to the original version are marked with commands (green). The code is written in Java but slightly simplified to improve readability.

```
// HDRF Select Partition
 public int selectPartition(Edge<Integer, Long> edge) {

      boolean madeup1 = false; // placeholder vertex 1
      boolean madeup2 = false; // placeholder vertex 2

      StoredObjectFixedSize first_vertex;
      StoredObjectFixedSize second_vertex;

    if (currentState.get(soureVertex)) {
        first_vertex =
            currentState.getRecord(edge.getSourceVertex);
      } else {
       // add placeholder v1
         first_vertex = new StoredObject();
         first_vertex.setDegree(1);
         madeup1 = true;
      }
      if (currentState.get(targetVertex)) {
         second_vertex =
            currentState.getRecord(edge.getTargetVertex);
      } else {
       // add placeholder v2
         second_vertex = new StoredObject();
```

```
          second_vertex.setDegree(1);
          madeup2 = true;
      }

      LinkedList<Integer> candidates = new
          LinkedList<Integer>();
    // for all partitions, calculate HDRF score
      for (int m = 0; m < k; m++) {
       // if "madeup" -> getDegree will return 1
          degree_u = first_vertex.getDegree();
       degree_v = second_vertex.getDegree();
       ...
          // all other calculations remain untouched
       // create candidate partitions
      }
      machine_id = candidates.get(choice);
      currentState.incrementMachineLoad(machine_id, e);
    // Degree Update - removed because degree
        information is updated by broadcat

    return machine_id;
  }
```

TRITA-EECS-EX-2019:558

www.kth.se