J. Log. Algebraic Methods Program. 139 (2024) 100966

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp

A computational framework based on the dynamic pipeline approach [☆]

Edelmira Pasarella [a,*], Maria-Esther Vidal [b,c,d], Cristina Zoltan [a], Juan Pablo Royo Sales [a]

[a] Computer Science Department, Universitat Politècnica de Catalunya, C/Jordi Girona, 1-3, 08034, Barcelona, Spain
[b] TIB-Leibniz Information Centre of Science and Technology, Hannover, Germany
[c] Leibniz University of Hannover, Hannover, Germany
[d] L3S Research Centre, Hannover, Germany

ARTICLE INFO

Dataset link: https://github.com/jproyo/dynamic-pipeline

Keywords:
Stream processing frameworks
Dynamic pipeline
Parallelism

ABSTRACT

Stream processing has inspired new computational approaches to facilitate effectiveness and efficiency. One such approach is the dynamic pipeline, which serves as a powerful computational model for stream processing. It is particularly well suited for solving problems that require incremental generation of results, making it an approach for scenarios where real-time analysis and responsiveness are critical. This paper aims to address a family of problems using the Dynamic Pipeline approach, and as a first step, we provide a comprehensive characterization of this problem family. In addition, we present the definition of a Dynamic Pipeline framework. To demonstrate the practicality of this framework, we present a proof of concept through its implementation and perform an empirical performance study. To this end, we focus on solving the problem of *enumerating or listing the weakly connected components* of a graph within the proposed framework. We provide two implementations of this algorithm to demonstrate the computational power and continuous behavior of the Dynamic Pipeline framework. The first implementation serves as a baseline for our experiments, representing an *ad hoc* solution based on the Dynamic Pipeline approach. In contrast, the second implementation is built on top of the developed framework. The observed results strongly support the suitability and effectiveness of the Dynamic Pipeline framework for implementing graph stream processing problems, especially those where continuous and real-time result generation is essential.

1. Introduction

In the ever-evolving landscape of data-intensive applications, dynamic data processing, characterized by the adaptive and responsive manipulation of large datasets in real time, and incremental generation of results, which involves continuously updating outcomes as new data becomes available, represent pivotal approaches. Efficiently addressing these facets is a critical challenge,

☆ Edelmira Pasarella is partially supported by MCIN/AEI/10.13039/501100011033 under grant PID2020-112581GB-C21. Maria-Esther Vidal is partially supported by Leibniz Best Minds: Programme for Women Professors project TrustKG-Transforming Data in Trustable Insights with grant P99/2020.

* Corresponding author.

E-mail addresses: edelmira@cs.upc.edu (E. Pasarella), maria.vidal@tib.eu (M.-E. Vidal), zoltan@cs.upc.edu (C. Zoltan), juan.pablo.royo@upc.edu (J.P. Royo Sales).

https://doi.org/10.1016/j.jlamp.2024.100966
Received 18 January 2022; Received in revised form 15 April 2024; Accepted 15 April 2024

Available online 22 April 2024

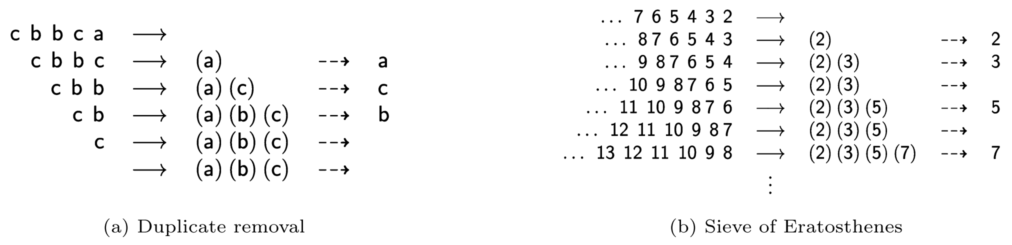


Fig. 1. Examples of dynamic pipelines. Values inside the parentheses correspond to data stored in the dynamic pipeline. Black arrows represent flow between input and stored data, while dashed arrows model the flow between the stored data and the produced output.

particularly when seeking fast and incremental results for big data algorithmic problems across diverse domains such as financial trading, healthcare monitoring, cybersecurity, supply chain management, telecommunications, and autonomous systems [11,22].

As an illustrative example, let us consider the problem of counting duplicates in a continuous stream of words, emphasizing the dynamic nature of real-time data processing. Further complexity is introduced when specific time intervals demand insights into which words have appeared in a fragment of the processed input stream and their respective frequencies. The finite yet unpredictable number of distinct words in the stream poses a challenge in allocating resources and determining system workload *a priori*. This inherent uncertainty underscores the necessity for an adaptive approach, dynamically reconfiguring system workload and resources. Further, the chosen approach should facilitate incremental generation of partial results.

This seemingly simple yet representative problem is pervasive across real-world scenarios requiring real-time processing. Consider, for instance, the task of monitoring and counting various types of vehicles on a highway, aiming to establish a real-time decision-making system for controlling heavy truck traffic when road density exceeds a predefined threshold. Data collected by sensors, assigning alphanumeric codes to each vehicle type 24/7, forms a continuous stream for system input. Realistic decision checkpoints are activated based on a predefined policy of observation time intervals. In this context, our work addresses the challenge of devising a computational model conducive to an adaptive approach and capable of incremental result generation.

Among the various techniques that have been explored to address dynamic data processing, parallel approaches that take full advantage of computational power have received considerable attention [5,17,19,37]. One such approach is the Dynamic Pipeline approach (DPA) [40], which has recently emerged as a prominent model for stream processing. This approach is based on *on-the-fly* pipeline parallelism [22], where a linear pipeline is dynamically constructed during execution.

To get an intuition about how the Dynamic Pipeline approach works let us consider two simple examples i) a simplification of the problem of counting duplicates: the duplicate removal problem and ii) finding all prime numbers by using the sieve of Eratosthenes algorithm. Fig. 1 depicts dynamic pipelines for solving these two problems. Black arrows represent the flow between the input data and the data stored in the pipeline; these data are enclosed in brackets. Dashed arrows represent the flow between the stored data and the output. Fig. 1a shows a simple example of duplicate removal where a dynamic pipeline receives as input a (finite) stream of characters `c b b c a` from which to remove duplicates. When the pipeline consumes a character from the stream, that character is stored and output, if the character has not been seen before. Otherwise, it is discarded. The pipeline stops when there are no more characters in the stream. Fig. 1b shows how prime numbers are filtered. When a prime number is detected, it is stored and output. Otherwise, it is discarded. In this case, the input stream of the dynamic pipeline is infinite. We can note in both examples two of the main features of dynamic pipelines: i) the incremental generation of results and ii) the way in which it increases as new data must be stored in the pipeline. Despite the simplicity of these examples, the expressiveness and generality of the Dynamic Pipeline approach can be observed.

In the pipeline parallelism approach, computations are divided into a series of *stages*. At each stage of the process, data from the previous stage serves as input and is then passed downstream to the next stage. The orchestration of a linear pipeline unfolds in the form of an iterative loop that continuously consumes input data. At each iteration of this loop, the ingested data item is subjected to processing. In particular, the parallelism inherent in this design is achieved by overlapping the execution of iterations over time. In essence, an iteration can start after its predecessor has started and before its own execution is mandated to complete. Typically, communication between stages takes place through channels. *Stages* in the Dynamic Pipeline approach are *stateful operators* [33]. A stateful operator stores received data elements as state. This state can be used and/or updated when a computation is performed. States are communicated through channels, in this computation pattern, the number of stages in a pipeline is not predetermined. Instead, pipelines dynamically adjust their size based on incoming data, allowing for flexible and efficient processing.

When applying the Dynamic Pipeline approach to problem-solving, two primary concerns come to the forefront. The first involves determining the suitability of this computational pattern for a given problem, necessitating a decomposition of the problem according to the Dynamic Pipeline computational pattern. We address this concern by providing a thorough characterization. The second concern revolves around the challenges inherent in implementing dynamic pipelines, primarily stemming from the absence of a dedicated tool tailored to this computational model. This often compels developers to construct the entire framework from scratch, diverting their efforts away from solving specific problems. Our objective is to alleviate this limitation by introducing a Dynamic Pipeline framework (DPF) that works analogously to how Hadoop helps developers use MapReduce. With the DPF, users only need to define the functionality of stages (in the form of scripts) and the characteristics of channels (including their types and connections). This arrangement shifts the responsibilities for data management, process spawning, and intricacies of parallelism to the system, offering users a more streamlined and focused problem-solving experience. However, a significant challenge in implementing a

Dynamic Pipeline Framework (DPF) lies in identifying an appropriate set of tools and a programming language capable of leveraging both of its primary aspects: i) *fast parallel processing* and ii) *theoretical foundations* that elevate computation to a first-class citizen.

Problem research and objectives. The main objective of this work is to characterize a family of problems to be addressed according to the Dynamic Pipeline approach, define a Dynamic Pipeline framework, and present a preliminary implementation of the framework. The aim of this implementation is to establish the particular features to be considered for an effective and efficient implementation of the DPF. Moreover, by focusing on a well-defined and highly pertinent problem, such as the enumeration or listing of weakly connected components (Weakly Connected Components (WCC)) within a graph, we can comprehensively evaluate the implications of using the developed framework to address challenges that require a progressive generation of results.

Proposed solution. We present a formal definition of the *dynamic pipeline approach* that provides the basis for implementing a robust tool based on the DP computational model. We explore the basics of implementing a DPF in a (parallel) pure functional language such as Haskell. That is, we determine the particular features (i.e., versions and libraries) that allow for an efficient implementation of a DPF. We also perform an empirical evaluation to analyze the performance of the dynamic pipeline framework implemented in Haskell. To evaluate the incremental generation of results, we measure the diefficiency metrics [1], i.e. the continuous efficiency of implementing a dynamic pipeline algorithm for enumerating weakly connected components incrementally on top of this framework with respect to an *ad hoc* Dynamic Pipeline solution.
Implementing the Dynamic Pipeline[1] requires importing the `DynamicPipeline` library. This concise implementation spans 27 source lines of code (SLOC), offering abstract methods for outlining the problem using the *dynamic pipeline approach*. Consequently, the `DynamicPipeline` library's methods enable a streamlined and comprehensible definition of generators, channels, states, and filters. On the other hand, the *ad hoc* implementation of the weakly connected graph components problem in Haskel[2] is much more complex. It requires the import of seven libraries, as well as the management of concurrency and parallelism. In total, about 200 lines of code are required to specify this *ad hoc* implementation.

Contributions. This paper extends the work reported by Royo-Sales et al. [34]. The contributions are as follows:

- Formalization of the key concepts associated with the Dynamic Pipeline approach.
- Characterization of problems suitable for the Dynamic Pipeline approach.
- Introduction of novel algorithmic templates for each stage within a dynamic pipeline.
- Definition and demonstration of the Dynamic Pipeline Framework (DPF), implemented in Haskell.
- Implementation of a dynamic pipeline algorithm for enumerating weakly connected components, based on the Dynamic Pipeline Framework.
- Empirical study comparing two dynamic pipeline implementations for incremental enumeration of weakly connected components in a graph. The first is an *ad hoc* dynamic pipeline approach, while the second corresponds to the previously mentioned implementation. The performance of these implementations is quantified using the diefficiency metrics introduced in [1].

The remainder of this paper is organized as follows. Section 2 presents the Dynamic Pipeline approach and its computational model. In addition, this section presents a characterization of the problems that can be solved using this approach. In Section 3, an exemplary Dynamic Pipeline algorithm is presented in detail: a DP for incrementally enumerating the weakly connected components of a graph. Next, Section 4 defines a generic dynamic pipeline framework and presents a proof of concept, an implementation in Haskell, highlighting the most important points to consider when developing and deploying a specific dynamic pipeline framework. Section 5 reports the results of the empirical evaluation of the implemented framework, considering two implementations of the algorithm for incrementally enumerating weakly connected components. Finally, Section 6 analyzes the related work and positions the Dynamic Pipeline Framework with respect to existing approaches and Section 7 summarizes our conclusions and outlooks on our future work, respectively.

2. The dynamic pipeline approach

This section serves as an introduction to the concepts that underpin the dynamic pipeline approach. Initially, dynamic pipelines are introduced and defined with a focus on their structure in terms of stages. Following this, the section provides a characterization of dynamic pipeline problems, exemplified by the specific challenge of counting duplicates in a stream of words and the clustering problem. The illustration aims to enhance understanding and highlight the practical application of dynamic pipelines. In the concluding part of this section, we present a computational model tailored specifically for dynamic pipelines. This model comprises basic elements necessary for effective implementation and underscores the theoretical foundations supporting the Dynamic Pipeline approach. This section describes these basic concepts, problem characteristics, and a computational framework that define the Dynamic Pipeline approach.

[1] https://hackage.haskell.org/package/dynamic-pipeline-0.3.2.0/docs/DynamicPipeline.html.
[2] https://github.com/jproyo/upc-miri-tfm/blob/17ee929f64a8be8a88ced782bfcf6bf355d8580a/connected-comp/src/ConnComp/Internal.hs.

2.1. A dynamic pipeline

A dynamic pipeline is conceptualized as an evolving directed multigraph, wherein vertices represent distinct stages and edges symbolize inter-stage communication channels. Essentially, the Dynamic Pipeline (DP) approach, as defined by [40], establishes a computational model rooted in the deployment of a linear pipeline. Within this framework, the utilization of *stateful operators* [33] becomes pivotal for modeling data behavior and transformations within the pipeline. Defined as computational entities, stateful operators encapsulate both a memory state and a function. Consequently, the stages of the pipeline align with stateful operators connected through communicating *channels*, synchronized by data availability. This subsection proceeds by introducing a formal model for a dynamic pipeline.
There are two types of stages in a dynamic pipeline:

- *Structural Stages*: Consisting of the Source (Sr), Generator (G) and Sink (Sk) stages, which are responsible for ingesting data, creating new functional stages, and outputting results, respectively. There is only one instance of each structural stage.
- *Functional Stages*: Consisting of different Filter (F) stages. Many functional stages can be responsible for processing the input data. A DP grows and shrinks depending on the spawning and lifetime of its functional stages.

Definition 1. Stages. Given a set of data types $\mathcal{T}$ and $\mathcal{F}$ a function space, $\mathcal{F} \subseteq \mathcal{T} \to \mathcal{T}$. A stage is an element st with type in the set $\{\mathsf{Source}, \mathsf{Filter}, \mathsf{Generator}, \mathsf{Sink}\}$.

- If the type of st is Source, then $st = (t_{Sr}, f_{Sr})$, where $t_{Sr} \in \mathcal{T}$ represents the type of the state of st, and $f_{Sr} \in \mathcal{F}$ is a function that transforms its input into a given format.
- If the type of st is Filter, then $st = (t^p_F, t_F, f_F)$, where $t_F \in \mathcal{T}$ represents the type of the state or memory of st and $t^p_F \in \mathcal{T} \cup \{\bot\}$ corresponds to the type of an optional filter parameter. $f_F \in \mathcal{F}$ is a function corresponding to the behavior of the Filter.
- If the type of st is Generator $st = (F, t_G, f_G)$, where F is a stage of type Filter, while $t_G \in \mathcal{T}$ and f_G represent the type of state of st, and a function specifying the behavior of the Generator stage.
- If the type of st is Sink, then $st = (t_{Sk}, f_{Sk})$, then $t_{Sk} \in \mathcal{T}$ designates the type of the state of st, and $f_{Sk} \in \mathcal{F}$ is the function characterizing the behavior of the Sink stage.

Definition 2. Instances of Stages. Given a set of data types $\mathcal{T}$ and $\mathcal{F}$ a function space, $\mathcal{F} \subseteq \mathcal{T} \to \mathcal{T}$. Let st be a stage in the set $\{\mathsf{Source}, \mathsf{Filter}, \mathsf{Generator}, \mathsf{Sink}\}$. An instance of st, denoted by $\overline{st}$, is defined as follows:

- If the type of st is Source, $st = (t_{Sr}, f_{Sr})$, then $\overline{st} = (v, f_{Sr})$, where v is an element in the type t_{Sr}, i.e., $v \in t_{Sr}$.
- If the type of st is Filter, $st = (t^p_F, t_F, f_F)$, then $\overline{st} = (v_1, v_2, f_F)$, where $v_1 \in t^p_F$ and $v_2 \in t_F$.
- If the type of st is Generator, $st = (F, t_G, f_G)$, then $\overline{st} = (F, v, f_G)$, where $v \in t_G$.
- If the type of st is Sink, $st = (t_{Sk}, f_{Sk})$, then $\overline{st} = (v, f_{Sk})$, where $v \in t_{Sk}$.

Example 1. Consider a Source stage, denoted as $st = (\mathsf{Int}, \mathsf{f})$. Instances of st can be represented as $\overline{st}_1 = (5, \mathsf{f})$ and $\overline{st}_2 = (0, \mathsf{f})$. In different instances of st, the value of the first component varies, while the function (i.e., the behavior) remains constant. Now, let us consider a Generator stage, denoted as $st' = (\mathsf{F}, \mathsf{Int}, \mathsf{g})$ where $\mathsf{F} = (t^p_F, t_F, f_F)$. Possible instances of st' are $\overline{st}'_1 = (\mathsf{F}, 0, \mathsf{g})$ and $\overline{st}'_2 = (\mathsf{F}, 7, \mathsf{g})$, illustrating variations in the first component while maintaining the same function. However, the 3-tuple $(\mathsf{F}, 0, \mathsf{h})$ is not a valid instance of st' for any $h \neq g$. In summary, this example illustrates the variability of stage instances, showcasing how different values can be assigned to components while preserving the consistent behavior defined by the associated functions.

Example 2. To illustrate the definition of stage specifications and instances within a Dynamic Pipeline (DP), consider the problem of counting duplicated words introduced in Section 1. Let us refer to the DP designed for this task as Dup. The stages of Dup are defined as follows:

- The stage of type Source is the pair (Sys, f^D_{Sr});
- The stage of type Filter is the pair $(\mathsf{String}, \mathsf{Int}, \mathsf{f}^D_F)$;
- The stage of type Generator is the pair $((\mathsf{String}, \mathsf{Int}, \mathsf{f}^D_F), \mathsf{Sys}, \mathsf{f}^D_G)$; and
- The stage of type Sink is the pair (Sys, f^D_{Sk}),

where Sys is the system status type, and f^D_{Sr}, f^D_F, f^D_G, and f^D_{Sk} are functions defining the behavior of the stages. Additionally, examples of $\mathsf{Filter}_{\mathsf{Dup}}$ instances are $('\mathsf{ABC}', 5, \mathsf{f}^D_F)$ and $('\mathsf{CDJ03}', 12, \mathsf{f}^D_F)$. This example shows the specifications and instances of stages within $\mathsf{DP}_{\mathsf{Dup}}$, providing a clear understanding of the types and associated functions stating the behavior of each stage.

In the context of dynamic pipelines, it is essential for structural stages such as Source and Sink to possess the capability to communicate with the external environment, facilitating the ingestion and output of information. To formalize our model of dynamic pipelines, we take a twofold approach. Initially, we extend the definition of a *pin graph*, as outlined in [20], to introduce the concept

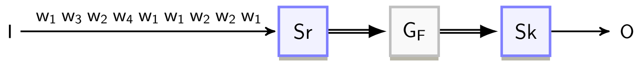


Fig. 2. A DP_0. Boxes represent stage instances Sr, G_F and Sk. Input and output pins are represented by single arrows. Channels are represented by double arrows meaning the existence of different types of channels between stages. Data arrive to the DP_0 through the input pin. The subindex F in G_F means that the filter specification F is parameter of generator stages.

of a *Dynamic Pipeline*. A *pin graph* is represented as a 5-tuple $G = (V, E, P, I, O)$, where (V, E) constitutes a directed graph, P is a finite set, and $I \subset P \times V$ forms the set of *input-pins* for G, while $O \subset V \times P$ constitutes the set of *output-pins* for G.

Definition 3. Dynamic Pipeline. Given a set of data types $\mathcal{T}$, a function space $\mathcal{F}$, $\mathcal{F} \subseteq \mathcal{T} \to \mathcal{T}$, and a set of stages $\{\mathsf{Source}, \mathsf{Filter}, \mathsf{Generator}, \mathsf{Sink}\}$. A *dynamic pipeline* dp is a 9-tuple defined as follows:

- $dp = (\mathbb{F}, \mathsf{Instances}, \mathsf{Chan}, \mathsf{Ins} \cup \mathsf{Outs}, \mathsf{I}, \mathsf{O}, \mathcal{T}, \mathsf{src}, \mathsf{tgt})$;
- $\mathbb{F}$ is a finite set of Filter instances;
- $(\mathsf{Instances} \cup \mathbb{F}, \mathsf{Chan})$ is a directed multigraph, where
  i) Instances is a set of three elements corresponding to the instances of stages of types Source (aka. $\overline{st_{Sr}}$), Generator (aka. $\overline{st_G}$), and Sink (aka. $\overline{st_{Sk}}$), i.e., $\mathsf{Instances} = \{\overline{st_{Sr}}, \overline{st_G}, \overline{st_{Sk}}\}$.
  ii) $\mathsf{Chan} = \{\mathsf{c} \mid \mathsf{c} = (\mathsf{id}, \mathsf{t}, \mathsf{q}) \wedge \mathsf{id} \in \mathsf{String} \wedge \mathsf{t} \in \mathcal{T} \wedge \mathsf{q} \in \mathsf{Queue(t)}\}$[3];
  iii) src, tgt are functions determining the endpoints of the edges, $\mathsf{src}, \mathsf{tgt} : \mathsf{Chan} \to \mathsf{Instances}$.
- Ins is a set of sources of input data;
- Outs is a set of recipient of results;
- $\mathsf{I} \subset \mathsf{Ins} \times \{\overline{st_{Sr}}\}$, called *input pins* of dp; and
- $\mathsf{O} \subset \{\overline{st_{Sk}}\} \times \mathsf{Outs}$, called *output pins* of dp.

Notice that identifying channels with an id allows for having more than one channel of the same type between two stages. In addition, even though Instances and $\mathbb{F}$ are sets, it is worth to stress that there are not two identical instances in a dynamic pipeline.

Definition 4. Initial Dynamic Pipeline. An *initial dynamic pipeline*, DP_0, is a dynamic pipeline composed of three instances of stages: Source, Generator, and Sink (see Fig. 2). Formally, DP_0 is defined as follows
$\mathsf{DP}_0 = (\emptyset, \mathsf{Instances}_0, \mathsf{Chan}_0, \mathsf{Ins} \cup \mathsf{Outs}, \mathsf{I}, \mathsf{O}, \mathcal{T}, \mathsf{src}, \mathsf{tgt})$, where $\mathsf{Instances}_0 = \{\overline{st_{Sr0}}, \overline{st_{G0}}, \overline{st_{Sk0}}\}$
$\mathsf{Chan}_0 = \{\mathsf{c} \mid \mathsf{c} = (\_, \_, \mathsf{emptyQueue})\}$[4] and the following condition holds:
i) For all $\mathsf{c} \in \mathsf{Chan}_0$, if $\mathsf{src(c)} = \overline{st_{Sr0}}$ then $\mathsf{tgt(c)} = \overline{st_{G0}}$ and if $\mathsf{src(c)} = \overline{st_{G0}}$ then $\mathsf{tgt(c)} = \overline{st_{Sk0}}$
ii) For all $\overline{S} \in \mathsf{Instances}_0$, the memory of $\overline{S}$ is empty, i.e., emptyMem.

Example 3. Consider an initial dynamic pipeline, denoted as DP_0, designed to address the challenge of identifying duplicate words, as illustrated in Fig. 2. The input stream of words is sourced from I, and the results are directed to a recipient O. In the visual representation, boxes correspond to instances of different stages, while arrows and thick arrows symbolize input/output pins and channels, respectively.

The ability of dynamic pipelines to evolve requires a mechanism that allows new instances of Filter instances to be added to pipelines. In addition, whenever an instance stage completes its task, it must be removed from the pipeline. In the Dynamic Pipeline approach, this mechanism is provided by two operators: the function spawn, for adding Filter instances, and the function die, for removing instances.

Definition 5. Evolving Operators. Let $\mathcal{DP}$ be a set of dynamic pipelines DP, such that $DP = (\mathbb{F}, \mathsf{Instances}, \mathsf{Chan}, \mathsf{Ins} \cup \mathsf{Outs}, \mathsf{I}, \mathsf{O}, \mathcal{T}, \mathsf{src}, \mathsf{tgt})$ where Instances is a set $\{\overline{st_{Sr}}, \overline{st_G}, \overline{st_{Sk}}\}$ of the instances of the set of Source, Generator and Sink, respectively. $\mathsf{Generator} = (F, t_G, f_G)$, $\mathsf{Filter} = (t^p_F, t_F, f_F)$, $t_G, t^p_F, t_F \in \mathcal{T}$ $\mathsf{p} \in t^p_F$, $\mathsf{m} \in t_F$ and $\overline{\mathsf{S}}$ is a stage instance. The operators spawn and die are the evolving operators of dynamic pipelines in $\mathcal{DP}$, which are defined as follows:

- $\mathsf{spawn} : \mathcal{DP} \times t^p_F \times t_F \to \mathcal{DP}$
  $\mathsf{spawn}((\mathbb{F}, \mathsf{Instances}, \mathsf{Chan}, \mathsf{Ins} \cup \mathsf{Outs}, \mathsf{I}, \mathsf{O}, \mathcal{T}, \mathsf{src}, \mathsf{tgt}), \mathsf{p}, \mathsf{m}) =$
  $(\mathbb{F}', \mathsf{Instances}, \mathsf{Chan}', \mathsf{Ins} \cup \mathsf{Outs}, \mathsf{I}, \mathsf{O}, \mathcal{T}, \mathsf{src}', \mathsf{tgt}')$, where
  $\mathbb{F}' = \mathbb{F} \cup \{(\mathsf{p}, \mathsf{m}, f_F)\}$
  $\mathsf{Chan}' = (\mathsf{Chan} \setminus \mathsf{Chan}_{\mathsf{Pred}}) \cup \mathsf{Chan}_\mathsf{L} \cup \mathsf{Chan}_\mathsf{R}$, where
  $\mathsf{Chan}_{\mathsf{Pred}} = \{\mathsf{c} \mid \mathsf{tgt(c)} = \overline{\mathsf{st}_\mathsf{G}}\}$,
  if $\mathsf{c} = (\mathsf{id}, \mathsf{t}, \mathsf{q}) \in \mathsf{Chan}_{\mathsf{Pred}}$ then

3 We assume the parameterized type Queue has defined the usual Abstract Data Type queue operators.
4 The underscore (_) stands for *any term* pattern as used in the programming languages Prolog and Haskell while emptyMem stands for the empty state/memory.

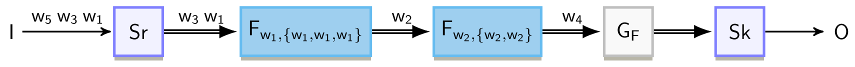


Fig. 3. A $\mathsf{DP_{DUP}}$ after creating two filter instances where the sub-indices $\mathsf{w_1, \{w_1, w_1, w_1\}}$ and $\mathsf{w_2, \{w_2, w_2\}}$, represent the parameters and the states of these instances, respectively. The sequences on the channel edges correspond to the queues of words. For the sake of not to overload the figure, we omit other details as the id and the type of channels.

i) $\mathsf{(id, t, q) \in Chan_L}$ and $\mathsf{src'((id, t, q)) = src(c) \wedge tgt'((id, t, q)) =(p, m, }f_F\mathsf{)}$
ii) $\mathsf{(id, t, emptyQueue) \in Chan_R}$ and $\mathsf{src'((id, t, emptyQueue)) = (p, m, }f_F\mathsf{)}$ and
$\mathsf{tgt'((id, t, emptyQueue)) = \overline{st_G}}$if $\mathsf{c \in Chan \setminus Chan_{Pred}}$ then $\mathsf{src'(c) = src(c)}$ and $\mathsf{tgt'(c) = tgt(c)}$

- die $: \mathcal{DP} \times (\mathsf{Instances} \cup \mathbb{F}) \rightarrow \mathcal{DP}$.
  $\mathsf{die((\mathbb{F}, Instances, Chan, Ins \cup Outs, I, O, \mathcal{T}, src, tgt), \overline{S}) =}$
  $\mathsf{(\mathbb{F}', Instances', Chan', Ins \cup Outs, I, O, \mathcal{T}, src', tgt')}$, where
  $\mathsf{\mathbb{F}' = \mathbb{F} \setminus \{\overline{S}\}}$
  $\mathsf{Instances' = Instances \setminus \{\overline{S}\}}$
  $\mathsf{Chan' = (Chan \setminus (Chan_{Pred} \cup Chan_{Suc})) \cup Chan_R}$, where
  $\mathsf{Chan_{Pred} = \{c \mid tgt(c) = \overline{S}\}}$,
  $\mathsf{Chan_{Suc} = \{c \mid src(c) = \overline{S}\}}$,
  if $\mathsf{c = (id, t, q) \in Chan_{Pred}}$ and $\mathsf{c' = (id, t, q') \in Chan_{Suc}}$ then
  $\mathsf{c'' = (id, t, concat(q', q)) \in Chan_L}$ and $\mathsf{src'(c'') = src(c)}$ and $\mathsf{tgt'(c'') = tgt(c')}$

The spawn operator creates a new filter instance $\overline{F} = \mathsf{(p, m, }f_F\mathsf{)}$. This new Filter instance is positioned between the instance of the Generator in a dynamic pipeline DP and its predecessor, establishing the necessary channel connections. This operation extends DP. The die operator removes the stage instance S and its outgoing arcs. The predecessor stages of S are connected to its successor stages, effectively reducing the size of DP.
In addition to the flow of data through channels, the above operators facilitate the dynamic evolution of DP, allowing it to adapt and respond to changing computational requirements.

Example 4. Fig. 3 depicts a $\mathsf{DP_{DUP}}$ for the problem of counting duplicates after spawning two filter instances.

Specifying an algorithmic solution as a DP corresponds to defining the behavior of each stage to solve the particular problem, as well as the number and type of channels connecting them. The deployment of a DP consists in setting up the initial configuration, i.e. a $\mathsf{DP_0}$. The activation or execution of a DP starts when a stream of data elements arrives at the Source stage of the $\mathsf{DP_0}$. During DP execution, the Generator stage spawns Filter instances according to the incoming data. The evolution of a DP continues as the stage Sr ingests data, G creates new stages as necessary, and passes to Sk the results produced (incrementally) by applying – in a sense – a composition of instances of the functions in the specification of the F stage. If the data stream is bounded, the computation ends when the lifetime of all stages of DP has expired, i.e. when all stage instances have died. Otherwise, if the data stream is unbounded, the DP remains active and results are output incrementally. The Fig. 4 shows the whole evolution of a small $\mathsf{DP_{DUP}}$.

2.2. Characterization of dynamic pipeline problems

In a broad sense, the Dynamic Pipeline approach is well suited to address problems where the input data can be logically grouped based on a defined *grouping property*. This ability to categorize input data is critical because the presence of duplicate elements within the input data implies the presence of an underlying multiset structure. Consequently, each distinct group of input data, as determined by the grouping relation, effectively corresponds to a *block of the partition* within the multiset [3,4]. Moreover, the approach uses a function designed to operate on (multi)subsets. By skillfully integrating the results obtained by applying this function to different blocks, a unified solution is obtained for the entire input data set. This methodical integration of results guarantees the effective solution of the problem over the entire input data set.

Counting Duplicates from Input Data. To illustrate the concept, let us consider again the DUP problem, i.e. the task of counting duplicate entries from an input stream of words, approached here as a *DP-Problem*. In this context, a fundamental grouping property emerges: the equality of elements within the stream. For example, for the input stream $\mathsf{w_1\ w_2\ w_2\ w_1\ w_1\ w_3\ w_1\ w_2\ w_4}$, the concept of equality leads to the following partitioned blocks: $\mathsf{\{w_1, w_1, w_1, w_1\}}$, $\mathsf{\{w_2, w_2, w_2\}}$, $\mathsf{\{w_3\}}$, and $\mathsf{\{w_4\}}$. The process involves defining a function that maps each block to its length. By methodically composing the results of this function applied to all blocks within the partition, we obtain the sequence $\mathsf{(w_1, 4)(w_2, 3)(w_3, 1)(w_4, 1)}$. This sequence reveals the desired result - a pair for each element from the input data, formed by itself and the number of times it occurs.

Clustering Problem. Now consider the *clustering problem* from the domain of data mining. This problem involves the grouping of data based on common characteristics. The clustering process involves partitioning the input data according to some *clustering criterion*, all without any prior information about the nature of the data set [2]. An example of a clustering algorithm is the k-means algorithm, which deals with the fundamental task of determining the optimal value of k beforehand.
Looking at the clustering problem through the lens of a *DP-problem*, we discover that the grouping property is consistent with the

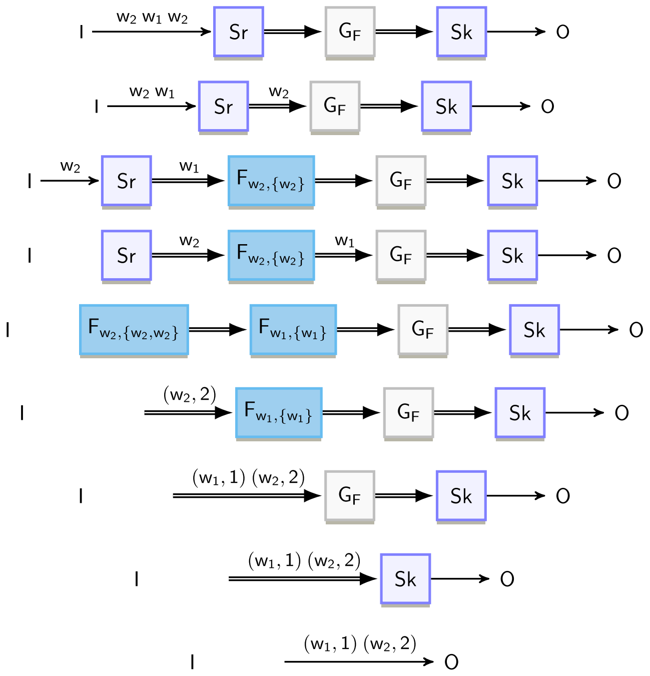


Fig. 4. Evolution of a DP_{DUP} with bounded input stream. Initial configuration is presented, followed by a Source stage ingesting data coming in at the input pin. The dynamic pipeline grows as new filter stages are created once a new word arrives at the Generator stage; the state changes in the filters. After receiving the last word in the input stream, the Source stage dies, filter stages send results to their output channels and also die, i.e., the DP starts decreasing as no new data arrives at the stages. Finally, results arrive at the output pin. Channels are added (removed) when the spawn function (or die function) is applied. There are two types of channels connecting stages, a channel of words and a channel of pairs, in a larger DP, both types of channels could transport data simultaneously.

clustering criterion - expressing similarity in a given context. The function applied to each partition block maps all its elements to an underlying multiset representing the corresponding cluster. Assembling these clusters in sequence yields the partition according to the clustering criteria. In addition, even in scenarios with unbounded data, it remains feasible to compute and deliver updated clusters as new data arrives.
This characterization of the clustering problem serves as an initial foundation for the development of an algorithm that competes with various iterations of the k-means algorithm, including the clustering of unbounded datasets. Although this challenge is beyond the scope of our current work, it remains an intriguing avenue for future research. Our research agenda specifically includes the exploration of this intriguing problem.

Definition 6. DP-problem. Let T, T' and T'' be sets of elements from countable domains $\mathbb{D}, \mathbb{D}'$ and $\mathbb{D}''$, respectively; they are not necessarily different. Let $\mathbb{P} = (I_{\mathbb{P}}, O_{\mathbb{P}})$ be a problem specification, $I_{\mathbb{P}} : Sequence(T) \to \mathsf{Bool}$ and $O_{\mathbb{P}} : Sequence(T) \times T' \to \mathsf{Bool}$, correspond to the pre and post conditions of $\mathbb{P}$, respectively. Elements in T are characterized by a *grouping property* $\mathcal{R}$. Let D be the multiset comprising the elements in $Sequence(T)$. $\mathbb{P}$ is a *DP-problem*, if and only if,

1. There exists a family of (multi)subsets $D_{\mathcal{R}} = \{B_i\}$, where $\forall i\, B_i$ is a block of the partition of the multiset D induced by $\mathcal{R}$
2. There exists a total function $F : D_{\mathcal{R}} \to T''$
3. There exists a function $C : (D_{\mathcal{R}} \to T'') \times 2^{D_{\mathcal{R}}} \to T'$

such that for every sequence $S \in Sequence(T)$ satisfying $I_{\mathbb{P}}$, $O_{\mathbb{P}}(S, C(F, D_{\mathcal{R}}))$ holds.

The following example refers to our running example on counting duplicates. In this problem, the grouping relation $\mathcal{R}$ is a binary relation that models equality. The induced partition by $\mathcal{R}$ corresponds to the quotient set of the equality relation. This is not the case in the clustering problem, where the clustering criterion could be based on different characteristics of the data, and therefore a data item could be assigned to different groups.

Example 5. Let $\mathbb{P}_1 = (I_{\mathbb{P}_1}, O_{\mathbb{P}_1})$ be the problem of counting duplicates from a sequence of words. The components are defined as follows:

- Input Data ($I_{\mathbb{P}_1}$): The input data S is structured as a bounded stream of words;

- Result ($O_{\mathbb{P}_1}$): The result is a sequence containing a pair (w, n) for each word w in the input sequence, where $n > 0$ corresponds to the number of occurrences of w;
- Grouping Property (Equality): The grouping property is the equality of words, i.e., $\forall w, w'$ occurring in S, if $w = w'$, then w and w' belong to the same block. This grouping property corresponds to $\mathcal{R}$ and induces $D_{\mathcal{R}}$;
- Function (F): The function $F :: D_{\mathcal{R}} \to words$, that represents the function that transforms a multiset comprising the same element into an integer, i.e., $F(\{w, \dots, w\}) = (w, length(\{w, \dots, w\}))$ for each block in $D_{\mathcal{R}}$; and
- Function (C): $C = map$, where $map(F, D_{\mathcal{R}})$ builds a sequence of words from the application of F to each element (block) of $D_{\mathcal{R}}$.

Using the Dynamic Pipeline approach to solve a problem $\mathbb{P}$ corresponds to characterizing it as a DP problem and then going into the concrete details of the definition, implementation, and use of the corresponding dynamic pipeline, $DP_{\mathbb{P}}$. Next, we present a computational model of the DP approach.

2.3. A computation model for the dynamic pipeline approach

The Dynamic Pipeline approach is based on both the dataflow model of computation and the pipeline parallelization model, as highlighted in previous work such as [22,30]. In particular, our approach builds on the recursive dynamic computation dataflow model. This model allows for the concurrent execution of multiple instances of a node and the on-the-fly creation of such nodes at runtime, as detailed in [14]. A typical DP computation evaluates the initial configuration and activation of the DP as input data begins to arrive, i.e., the parallel and asynchronous execution of each *executable stage*. A stage is executable if it has enough data to evaluate its function. The computation of the DP continues as long as there are executable stages. Note that the state/memory of instances of stages can change during the computation.
Specifying the four types of stages that make up a dynamic pipeline and what dynamic pipelines are is not enough to define the dynamic pipeline model of computation. Next, we present a generic operational DP model of computation, similar to how the von Neumann model of computation is characterized by the *fetch-decode-execute cycle*. [36,39] and the MapReduce programming model is introduced [18,19] by defining the parallel execution of functions MAP-SHUFFLE-REDUCE. In what follows, we present the Dynamic Pipeline model of computation in the form of an algorithmically structured approach that includes i) a *window policy* and ii) four algorithmic templates corresponding to concurrent/parallel processes - the Source, Filter, Generator, and Sink stages.

Window policy. According to stream processing techniques, results can be generated by processing sub-streams. This is according to a *window-based* approach [33,25]. A windowing policy can be based on different parameters that allow the processing of sub-streams (see [10]). Very well-known parameters are the window *scope*, i.e., a window size (e.g., a number of data elements), which allows to define *count-based policy*, and a window *slide*, i.e., data elements within a time interval, which allows to define *time-based policy*. These parameters define when to start a new window on the input stream. Deciding which type of window to use to define the window policy depends on the nature of the problem and probably on the (physical) system configuration. For implementing dynamic pipelines that handle unbounded input streams, a window policy is a key consideration, although it can also be applied to problems with bounded input streams.

Generic templates. The generic templates establish a consistent format and framework that greatly simplify the implementation of DP problems. Detailed in Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4, these templates are presented in a pseudo-code format. Notable statements in this pseudocode are i) The operator $\ll$ for sending to a channel (when channels are on the left) and receiving from a channel (when channels are on the right). We can think of channels as having input and output ports. Send and receive operations are blocked until the channel is ready. ii) Get allows to receive data from sources outside the DPs, i.e. from input pins, and Put allows to send values outside the DPs, i.e. to output pins. These kinds of operators mentioned above, suitable for managing pipelines of stages processing stream data, are available for instance, in Go language[5] and Elixir Flow.[6] iii) The control structure for( )do{...} corresponds to an infinite loop whose sequence of instructions is repeated indefinitely until a break instruction, if any, is reached. A break statement passes control to the statement following the loop in the immediate outer block (see RE_1 control structure in [21]. Without loss of generality, input data is assumed to be an unbounded data stream. It is assumed that $DP_{\mathbb{P}}$ operates within an abstract machine with infinite resources (memory, processors, etc.).
The generic Source template is guided by a window policy, ϑ, defined in terms of the state of the system. In fact, the policy ϑ effectively manages the ingestion of sub-streams and, in addition, a signal, ρ_1, is used to notify the rest of the DP to stop ingesting data from their input channels and start collecting the results produced by the processing of the data ingested during the last window. If aggregated results need to be computed, an initial value, v_0, is provided to the rest of the DP.
Within Algorithm 1, data ingestion adheres to the specified window policy ϑ (Lines 2-4). In Algorithm 2, CreateGroup() (Lines 1-17) orchestrates the construction of partition blocks $D_{\mathcal{R}}$ linked to filter instances. The decision function $\mathcal{V}_{\mathcal{R}}$ is utilized to determine data item eligibility as a block (Lines 5-7). Furthermore, result emission is carried out through the EmitResults() function (Lines 18-31). In Line 25, we can see how function F is applied to the block stored in the filter stage. In particular, ρ_1 governs the cessation of group creation and results' generation (Line 17). CreateGroup() and function F operate within the filter's executable context.
Algorithm 3 manages data ingestion and the creation of new filter instances until the ρ_1 signal is received. Depending on the problem,

[5] https://go.dev/doc/effective_go#concurrency.
[6] https://hexdocs.pm/flow/Flow.html.

$\mathsf{Param}_{\mathbb{P}}$ and $\mathsf{InitialBlock}_{\mathbb{P}}$ functions dictate the initial values bound to the parameters of spawn (Line 5). Partial results are passed to the subsequent stage (Sink) (Lines 10-17). It must be kept in mind that, when considering real machines, it is necessary to deal with limited resources. Information about resource consumption should be tracked in System State logs. Among others, the Generator stage must use this information for determining whenever a new Filter instance can be created (spawn) by checking its state $\mathsf{M}_{\mathsf{G}_{\mathbb{P}}}$. Since we are considering an abstract machine with infinite resources in this paper, these details are not considered in Algorithm 3.
Finally, Algorithm 4 is responsible for applying function C to the results computed by F in Algorithm 2. The function C can be an aggregation operator, e.g., a Haskell-like fold operator, but it can also be a Python-like map, i.e., a function that works as an iterator to return results after applying the function F to blocks in filters. In Line 5, the $\uplus$ operator is used to construct the real parameter of the function C. In Line 10 the result is sent to the output pin. Observe how the function C is applied in the second argument of the function Put. We assume that the state of the $\mathsf{Sk}_{\mathbb{P}}$ stage is initially set to a convenient value and is reset to this initial value after the Put instruction in Line 5.
In Algorithm 2, the creation of blocks within the partition $\mathcal{D}_{\mathcal{R}}$ associated with filter instances is facilitated by CreateGroup(). While the presented algorithm implies that blocks are constructed in a single round, as seen in scenarios like the counting duplicates problem, a more complicated reality emerges. Taking the example of counting duplicates, the task remains consistent regardless of whether the input data stream is boundless or finite. The initial occurrence of a word in the input stream alone prompts the establishment of its corresponding block within the input data stream's partition. Consequently, the emission of results via the EmitResults() function can commence.
However, the generality of this process becomes apparent when considering diverse DP-problems. In such cases, the block construction, executed by CreateGroup(), iteratively execute several creating group processes, which could be seen as a composition of functions, i.e., $\mathsf{CreateGroup} = \mathsf{cg}_{\mathsf{r}} \circ \cdots \circ \mathsf{cg}_1$. This characterization designates $\mathbb{P}$ as a *DP-problem of* r *rounds*, where each successive round refines block construction and may yield intermediate data types. It is important to highlight that demarcating the boundary between CreateGroup() and EmitResults() is not always a straightforward endeavor. The subsequent section presents an illustrative example.

Algorithm 1: Generic Source Stage ($\mathsf{Sr}_{\mathbb{P}}$).

```
   Input     : S of type Sequence(T)
   Channels: ⟨$\mathsf{C_T}$ of type $\mathsf{T} \cup \{\rho_1\}$, $\mathsf{C_{T''}}$ of type $\mathsf{T''} \cup \{\rho_1, \mathsf{v}_0\}$⟩
   State     : $\mathsf{M}_{\mathsf{Sr}_{\mathbb{P}}}$ of type Sys
   Script$_{\mathsf{Sr}_{\mathbb{P}}}$ :
1  for (  ) do
2      while ($\vartheta(\mathsf{M}_{\mathsf{Sr}_{\mathbb{P}}})$) do
3          $\mathsf{C_T} \ll \mathsf{Get(S)}$
4      end
5      $\mathsf{C_T} \ll \rho_1$;
6      $\mathsf{C_{T''}} \ll \mathsf{v}_0 \ll \rho_1$;
7  end
```

Note that the instances of the stages defined by the σ function in Section 2 correspond to the different stages reached by the channels, computations, and states specified in each template.

Proposition 1. *Let $\mathbb{P}$ denote a DP-problem structured as $\mathbb{P} = (I_{\mathbb{P}}, O_{\mathbb{P}})$, where $I_{\mathbb{P}} : Sequence(T) \to \mathsf{Bool}$ and $O_{\mathbb{P}} : Sequence(T) \times T' \to \mathsf{Bool}$ represent the pre- and post-conditions of $\mathbb{P}$, respectively. The elements within the set T are defined by a distinctive property referred to as the* grouping property *$\mathcal{R}$. Suppose D constitutes a multiset that encompasses the elements found in $Sequence(T)$, partitioned according to the characteristic $\mathcal{R}$, denoted as $D_{\mathcal{R}}$. Within this context, let $F : D_{\mathcal{R}} \to T''$ constitute a total function, and let $C : (D_{\mathcal{R}} \to T'') \times 2^{D_{\mathcal{R}}} \to T'$ be a higher-order function such that for every sequence $S \in Sequence(T)$ satisfying $I_{\mathbb{P}}$, $O_{\mathbb{P}}(S, C(F, D_{\mathcal{R}}))$ holds. If F and C are computable functions, and the problem of determining if $\mathcal{R}$ is satisfied is decidable, then there exists a computational solution using the dynamic pipeline pattern. As a result of this setup, $DP_{\mathbb{P}}$ will eventually reach the* Sink *stage (reflecting liveness) and produce the results (indicating progress).*

Proof sketch. The computability of functions F and C, along with the decidability of $\mathcal{R}$ satisfiability, guarantees that at each stage of $DP_{\mathbb{P}}$, well-defined operations are carried out within finite time. This ensures that $DP_{\mathbb{P}}$ will reach a termination point, and Algorithm 4 will be executed. The correctness of the dynamic programming process relies on the consistency of F and C in capturing the essence of the DP-problem. With these computable functions, the process incrementally refines its calculations, accurately incorporating relevant information while adhering to the dynamic programming principles.
The decidable nature of $\mathcal{R}$ satisfiability allows $DP_{\mathbb{P}}$ to accurately identify and partition elements based on the grouping property $\mathcal{R}$. This ensures that the data is suitably processed, grouped, and directed towards the appropriate stages.
Consequently, the execution of Algorithm 4 based on C ensures the generation of accurate results, as $O_{\mathbb{P}}(S, C(F, D_{\mathcal{R}}))$ holds for sequences satisfying $I_{\mathbb{P}}$. Moreover, the adherence to the window-policy ϑ guarantees the appropriate handling of the processed data streams, culminating in the desired results.
A formal proof of this proposition would require the concrete details of the window policy (ϑ) as well as the used data structures, the implementation of the functions $\mathcal{VR}$, F and C, the implementation of Algorithms 1, 2, 3 and 4 and, possibly, some specific details

Algorithm 2: Generic Filter Stage ($F_{\mathbb{P}}$).

```
Parameter: $\hat{x}$ of type $T \cup \{\bot\}$
Channels : $\langle C_T$ of type $T \cup \{\rho_1\}, C_{T''}$ of type $T'' \cup \{\rho_1, v_0\}\rangle$
State    : $M_{F_{\mathbb{P}}}$ of type block(T)
def CreateGroup()
    for ( ) do
        $x \ll C_T$
        if ($x != \rho_1$) then
            if ($\mathcal{V}_{\mathcal{R}}(x, M_{F_{\mathbb{P}}})$) then
                # Custom code for updating block here
            else
                # item x must be passed to the next stage
                $C_T \ll x$
            end
        else
            break
        end
    end
    # $\rho_1$ has been read from channel $C_T$ and thus, stop
    # ingesting data from it and pass this signal to next stage
    $C_T \ll \rho_1$
def EmitResults()
    $r \ll C_{T''}$
    while ($r != \rho_1$) do
        # Notice that $v_0$ has been read from channel $C_{T''}$
        # Custom code involving r here or may be just $C_{T''} \ll r$
        $r \ll C_{T''}$
    end
    $C_{T''} \ll F(M_{F_{\mathbb{P}}})$
    $C_{T''} \ll \rho_1$
    # results generated in this filter instance have been passed to the next stage followed by $\rho_1$
Script$_{\mathbb{P}}$ :
for ( ) do
    CreateGroup()
    EmitResults()
end
```

Algorithm 3: Generic Generator Stage ($G_{\mathbb{P}}$).

```
Parameter: F of type $F_{\mathbb{P}}$
Channels : $\langle C_T$ of type $T \cup \{\rho_1\}, C_{T''}$ of type $T'' \cup \{\rho_1, v_0\}\rangle$
State    : $M_{G_{\mathbb{P}}}$ of type Sys
Script$_{G_{\mathbb{P}}}$ :
for ( ) do
    for ( ) do
        $x \ll C_T$
        if ($x != \rho_1$) then
            spawn(F, Param$_{\mathbb{P}}$(x), InitialBlock$_{\mathbb{P}}$(x))
        else
            break
        end
    end
    for ( ) do
        $r \ll C_{T''}$
        if ($r != \rho_1$) then
            $r \ll C_{T''}$
        else
            break
        end
    end
    $C_{T''} \ll \rho_1$
end
```

of the system. In particular, we think that this kind of proof can be done using formal techniques based on some extensions of linear temporal logic such as the event-driven linear temporal logic presented in [9].

Algorithm 4: Generic Sink Stage ($\mathsf{Sk}_{\mathbb{P}}$).

```
Output   : R of type T'
Channels: ⟨$C_{T''}$ of type T'' ∪ {$\rho_1$, $v_0$}⟩
State    : $M_{Sk_P}$ of type T''
Script$_{Sk_P}$ :
for (   ) do
    for (   ) do
        r ≪ $C_{T''}$
        if (r != $\rho_1$) then
            $M_{Sk_P}$ = r ⊎ $M_{Sk_P}$
        else
            break
        end
    end
    Put(R,$\mathcal{C}$($M_{Sk_P}$))
end
```

3. A DP algorithm for the weakly connected components

Consider the task of enumerating the *Weakly Connected Components* (WCC) of a given graph $G = (V, E)$ using the DP approach. Given G with no isolated vertices or loops, the challenge of computing its weakly connected components is a well-studied problem in graph theory. Weakly connected components are defined without regard to the orientation of edges in a graph, if any. A weakly connected component denotes a maximally connected subgraph within G, meaning that there exists a path connecting every pair of vertices in G [12]. Obviously, the connectivity relation between vertices is reflexive, symmetric, and transitive, making it an equivalence relation. Consequently, the task of *computing* weakly connected components in G is to compute the quotient set of V based on this connectivity relation. More simply, it involves computing the finest partition $cc_1, \ldots, cc_k$ of V such that for every $i \in [1 : k]$, $cc_i \subseteq V$, and for every distinct $v, w \in cc_i$, v and w are connected. The subgraphs induced by each subset $cc_i \subseteq V$ represent the *weakly connected components* of G. Consequently, finding the connected components of G naturally leads to deriving the partition of V driven by the connectivity relation.
An *incremental enumeration* approach to the WCCs of G specifically involves listing these components in a way that is consistent with their computation or identification process. We proceed by outlining the characterization of the problem of enumerating weakly connected components in a graph, as per Definition 6. This example encapsulates the essential components of the weakly connected component enumeration problem, illustrating its structure and key attributes.

Example 6. Consider $\mathbb{P}_2 = (I_{\mathbb{P}_2}, O_{\mathbb{P}_2})$, which represents the task of enumerating the weakly connected components of a given graph $G = (V, E)$. The components are defined as follows:

- Input data ($I_{\mathbb{P}_2}$): The input data S is structured as a bounded stream of edges of the form (v, w), where both v and w belong to V, the set of vertices in the graph.
- Result ($O_{\mathbb{P}_2}$): The result is a sequence containing the weakly connected components of G.
- Grouping Property (Connectivity): The grouping property centers around the concept of connectivity, $\leftrightarrow$ (i.e., the grouping property $\mathcal{R}$), such that for any v and w within V, if $v \leftrightarrow w$, then v and w are part of the same grouping block.
- Function (F): The function $F : D_{\mathcal{R}} \to 2^V$ is defined as $F(b) = b$, mapping a connection block b to itself.
- Function (C): $C = map$, where $map(F, D_{\mathcal{R}})$ builds a sequence of sets of vertices (the connected components) from the application of F to each element (block) of $D_{\mathcal{R}}$.

The Dynamic Pipeline approach is employed to facilitate the incremental enumeration of weakly connected components (WCCs) within a graph G, denoted as $\mathsf{DP}_{\mathsf{WCC}}$. To process this, a stream of (non-oriented) edges, culminating in a designated symbol eof, is input. The $\mathsf{DP}_{\mathsf{WCC}}$ configuration revolves around the distinctive behaviors exhibited by its four primary stage types: Source ($\mathsf{Sr}_{\mathsf{WCC}}$), Generator ($\mathsf{G}_{\mathsf{WCC}}$), Sink ($\mathsf{Sk}_{\mathsf{WCC}}$), and Filter ($\mathsf{F}_{\mathsf{WCC}}$) stages. The linear and unidirectional connections between these stages are facilitated through the channels $\mathsf{C_E}$ (for edge transmission) and $\mathsf{C_{set(V)}}$ (for transmission of sets of connected vertices). The functionality of the Source stage ($\mathsf{Sr}_{\mathsf{WCC}}$) is rooted in the identity transformation applied to the incoming stream of edges. As each edge enters, it is transmitted through $\mathsf{C_E}$ to the subsequent stage. Upon receiving the eof signal via $\mathsf{C_E}$, the Source stage ($\mathsf{Sr}_{\mathsf{WCC}}$) terminates. Algorithm 5 provides the definitive outline for the Source stage ($\mathsf{Sr}_{\mathsf{WCC}}$). Notably, the script $\mathsf{Sr}_{\mathsf{WCC}}$ (or $\mathsf{ScriptSr}_{\mathsf{WCC}}$) executes the operation f_{Sr} for source stages, as stipulated in Section 2.1. The loop from Lines 1 to 8 orchestrates the inclusion of input data into the edge channel $\mathsf{C_E}$, with $\ll$ denoting the insertion action. Line 11 is responsible for initializing the channel $C_{set(V)}$, with $\emptyset$ serving as the initial entry ($\mathsf{v_0}$ in the generic templates). The algorithm persists until the channel $C_{set(V)}$ concludes its operation and then, the $\mathsf{Sr}_{\mathsf{WCC}}$ stage dies. Roughly speaking, algorithm $\mathsf{DP}_{\mathsf{WCC}}$ (Algorithm 6) requires two rounds to enumerate the weakly connected components. In the $\mathsf{DP}_{\mathsf{WCC}}$, we observe that only Filter instance stages need to maintain a state. According to the definition of the weakly connected components, blocks in filters are represented as sets of vertices (SetOf(V)). In addition, since the input data stream is bounded, there is not necessarily a window-policy, but there is a mark to indicate the end of ingesting data items, the eof mark. Also, a function for closing channels, close(.) is used as well as the die(.) function. Besides, in the Filter template, as we anticipated

Algorithm 5: Source Stage ($\mathsf{Sr}_{\mathsf{WCC}}$).

```
Input    : S of type Sequence(V × V)
Channels : ⟨C_E of type E, C_SetOf(V) of type SetOf(V)⟩
Script_Sr_WCC :
1  for ( ) do
2      e ≪ Get(S)
3      if (e != eof) then
4          C_E ≪ e
5      else
6          break
7      end
8  end
9  C_E ≪ eof
10 close(C_E) ;
11 C_set(V) ≪ ∅ ;
12 close(C_set(V));
13 die(self)
```

in the previous section, the creation of blocks requires more than one round. There is no clear difference between the creation of the blocks and the generation of results. In the first round, filter instance stages receive the edges of the input graph and create sets of connected vertices (Lines 1-16). During the second round, each filter instance receives sets of connected vertices (Lines 19-36). When the incoming set of vertices intersects with its set of connected vertices in a filter, the set of incoming vertices is connected to the latter, and no output is produced. Otherwise, the incoming set of connected vertices is passed to the next stage, i.e., the behavior of $\mathsf{F}_{\mathsf{WCC}}$ is stated by the sequence formed by Connectivity followed by ConnectedComponents in Lines 39-40. Connectivity keeps a set of connected vertices in state $M_{\mathsf{F}_{\mathsf{WCC}}}$. When an edge e arrives, if one endpoint of e is present in the state, then the other endpoint of e is also added to $M_{\mathsf{F}_{\mathsf{WCC}}}$. Edges without incident endpoints are passed to the next step. If the channel C_{E} has no more edges, it is closed and ConnectedComponents starts its execution. When ConnectedComponents receives a set of connected vertices cc in $\mathsf{C}_{\mathsf{set(V)}}$, it determines whether the intersection between cc and the vertices in its state is not empty. If so, it adds the vertices in cc to the state $M_{\mathsf{F}_{\mathsf{WCC}}}$. Otherwise, cc is passed to the next step. If channel $\mathsf{C}_{\mathsf{set(V)}}$ is empty, ConnectedComponents passes –through $\mathsf{C}_{\mathsf{set(V)}}$– the set of vertices in its state to the next stage (Line 23 in Algorithm 6). ConnectedComponents closes $\mathsf{C}_{\mathsf{set(V)}}$ channel, and then the filter instance dies.
Algorithm 7 implements the generator stage. In the first round (Lines 1-9), a new instance is created, i.e., the operator $\mathsf{spawn}(\mathsf{F}_{\mathsf{WCC}}, \emptyset, \{\mathsf{v}, \mathsf{w}\})$ is executed. Additionally, Algorithm 7 applies the identity transformation to the entire set of vertices in channel $C_{\mathsf{SetOf(V)}}$; this is specified in Lines 11-17. Lastly, all the channels are closed and the generator dies (Lines 19-21).
Algorithm 8 implements the sink stage; it receives the identified weakly connected components (Lines 1-8) according to the Put(., .) function. The channel is closed (Line 9) and the sink dies (Line 10).

4. The dynamic pipeline framework

Solving problems using the Dynamic Pipeline approach is an intuitive task for some families of problems. However, due to the underlying concurrency/parallelism, implementing DPs from scratch requires high programming skills. In this section, we first define a generic Dynamic Pipeline framework and discuss the features of this definition. A Dynamic Pipeline framework is expected to help users create and deploy dynamic pipelines without being an expert in concurrency or parallelism. Thus, users only need to focus on specifying DPs and programming the functionality for their stages. Second, we present a proof of concept. Specifically, we implement a dynamic pipeline to enumerate weakly connected components ($\mathsf{DP}_{\mathsf{WCC}}$) using a preliminary implementation of a DPF using Haskell as the programming language. Haskell was chosen because functions are first-class citizens in this language. Therefore, we expected that this programming language would be suitable to naturally represent stateful operators when extended with some state management libraries. Finally, through an empirical evaluation of $\mathsf{DP}_{\mathsf{WCC}}$, we assess the performance of the Dynamic Pipeline framework developed in Haskell and thus the feasibility of this tool.

Definition 7. A *Dynamic Pipeline framework*, $\mathsf{DPF}_{\mathsf{L}}$, is a 5-tuple $\langle \mathsf{IDE}, \mathsf{XE}, \mathsf{IS}, \mathsf{OS}, \mathsf{R} \rangle$. L is the *Target Programming Language*, i.e., the language in which a dynamic pipeline DP will be implemented. The language L should support concurrency and parallelism. The components of $\mathsf{DPF}_{\mathsf{L}}$ are defined as follows:

1. IDE is an *Integrated Development Environment*; it is a user interface suitable for both expert and non-expert developers to create DP. The IDE consists of: i) An editor based on the stage templates presented in Section 7, written in the L language. This editor allows writing L-code oriented to customize the stage templates of DP according to the application; ii) A *Channel Type Checker* component, (T_{SL}), that verifies the consistency of data types in the users' customized stage's templates with respect to the input and output data in stages; iii) A DP *Assembler*, A_{S}. This component of $\mathsf{DPF}_{\mathsf{L}}$ generates the corresponding L-code, ready to be executed. To be concrete, A_{S} generates the initial configuration of DP; and iv) A repository for saving all the dynamic pipelines developed using the framework.

Algorithm 6: Filter Stage (F_{WCC}).

```
Parameter: ⊥
Channels : $\langle C_E$ of type E, $C_{SetOf(V)}$ of type SetOf(V)$\rangle$
State    : $M_{F_{wcc}}$ of type SetOf(V)
def Connectivity ( )
    for (   ) do
        $e \ll C_E$;
        if (e != eof) then
            $(v, w) = e$;
            if ($\{v, w\} \cap M_{F_{wcc}} \neq \emptyset$) then
                $M_{F_{wcc}} = M_{F_{wcc}} \cup \{v, w\}$ ;
            else
                $C_E \ll e$;
            end
        else
            break
        end
    end
    $C_E \ll$ eof;
    close($C_E$);

def ConnectedComponents ( )
    for (   ) do
        $cc \ll C_{set(V)}$;
        if (cc != eof) then
            if ($cc \neq \emptyset$) then
                if ($cc \cap M_{F_{wcc}} \neq \emptyset$) then
                    $M_{F_{wcc}} = M_{F_{wcc}} \cup cc$;
                else
                    $C_{set(V)} \ll cc$;
                end
            end
        else
            break
        end
    end
    $C_{set(V)} \ll M_{F_{wcc}}$;
    $C_{set(V)} \ll$ eof;
    close($C_{set(V)}$);

Script$_{F_{wcc}}$ :
Connectivity();
ConnectedComponents();
die(self)
```

2. XE is the DP *Execution Engine*; it is a process that controls the execution, i.e. starting, stopping, pausing, monitoring, etc., of dynamic pipelines.
3. IS is the *Input Data Source Manager System*. The source of input data could be sensors, file systems, IoT devices, etc.
4. OS is the *Output Data Recipients Manager*. Output data could be files, plots, signals, etc.
5. R is the *Fail Recovery System* that monitors, logs, and stores snapshots of DPF_L system status and activity. It is responsible for applying the recovery policies defined by DPF_L when necessary.

4.1. $DPF_{Haskell}$*: an implementation of the dynamic pipeline specification*

In this subsection, we describe the implementation of $DPF_{Haskell}$; it allows us to understand and explore the limitations and challenges that might be found in the development and deployment of a specific DPF. $DPF_{Haskell}$ is useful to provide general insights into the main features to consider when implementing a dynamic pipeline framework, and to demonstrate the feasibility of developing and using such a framework. In the following, we present the most important aspects we dealt with. All the specific details of the implementation can be found in [34].

Stream processing. Several implementations for streaming processing models[7][8][9] in Haskell have emerged over the years. These libraries have their abstractions and can do stream processing in a fast way with varying performance according to recent bench-

[7] https://hackage.haskell.org/package/conduit.
[8] https://hackage.haskell.org/package/pipes.
[9] https://hackage.haskell.org/package/streamly.

Algorithm 7: Generator Stage (G_{WCC}).

```
Parameter: $F_{WCC}$
Channels : $\langle C_E$ of type $E, C_{SetOf(V)}$ of type $SetOf(V)\rangle$
Script$_{G_{WCC}}$ :
1  for ( ) do
2      $e \ll C_E$
3      if ($e\,!= eof$) then
4          $(v, w) = e$;
5          spawn($F_{WCC}, \emptyset, \{v, w\}$);
6      else
7          break
8      end
9  end
10 close($C_E$);
11 for ( ) do
12     $cc \ll C_{set(V)}$
13     if ($cc\,!= eof$) then
14         $C_{set(V)} \ll cc$;
15     else
16         break
17     end
18 end
19 $C_{set(V)} \ll eof$;
20 close($C_{set(V)}$);
21 die(self)
```

Algorithm 8: Sink Stage (Sk_{WCC}).

```
Channels : $\langle C_{SetOf(V)}$ of type $SetOf(V)\rangle$
Output   : R of type Sequence(SetOf(V))
Script$_{Sk_{WCC}}$:
1  for ( ) do
2      $cc \ll C_{set(V)}$
3      if ($cc\,!= eof$) then
4          Put(R, cc);
5      else
6          break
7      end
8  end
9  close($C_{set(V)}$);
10 die(self)
```

marks.[10] Although they seem to be suitable for implementing a DP, it is necessary to know the disposition of the pipeline stages beforehand, and thus it is hard to achieve a concise and expressive implementation of a DPF. Moreover, since they are designed according to the data-parallel streaming model [33], implementing DP using these tools becomes a counterintuitive and difficult task. Another problem is that in these libraries it is not clear how to manipulate the channels between the stages to control the flow of data. We have developed a package that handles stream processing in the way that the dynamic pipeline model does. To the best of our knowledge, no similar library has been written in Haskell. In this implementation we have utilized an *Embedded Domain Specific Language* (*EDSL*) approach [8], embedded in the target language, i.e., Haskell to exploit the purely functional nature of that language. Accordingly, users write their dynamic pipeline specifications in Haskell, restricted to the EDSL abstractions. This approach allows the use of the Haskell's strong type system to provide type-level correctness [15].

Parallelization. One of the most crucial tasks of the implementation is the selection of concurrency libraries to support an intensive parallelization workload. Parallelization techniques and tools have been intensively studied and implemented in Haskell [29]. Indeed, it is well known that green threads and sparks allow one to spawn thousands to millions of parallel computations. These parallel computations do not penalize performance when compared with Operating System (OS) level threading [27]. In particular, we spawn green threads in this first version of the $DPF_{Haskell}$. Moreover, to spawn asynchronous computations [27] using green threads, we use the `async` library[11] [26]. This library provides combinators to manage thread terminations and errors. We also explored the possibility of using the concurrency library `Par` *Monad*, based on sparks [28]. Even though both, green threads and sparks, allow for spawning thousands to millions of parallel computations, the library `Par` *Monad* was designed for data parallelism (i.e. data are divided into disjoint subsets that are processed independently) and hence, it is practically impossible to do fine level of control on

10 https://github.com/composewell/streaming-benchmarks.
11 https://hackage.haskell.org/package/async.

Table 1
Datasets. These graphs compose the benchmark SNAP [24] and cover several characteristics, i.e., size, density, and number and size of WCC. The Google Web Graph is the most complex graph regarding all these parameters.

Network	vertices	Edges	Diameter	#WCC	#vertices Largest WCC
Enron Emails	36,692	183,831	11	1,065	33,696 (0.918)
Astro Physics Collaboration Net	18,772	198,110	14	290	17,903 (0.954)
Google Web Graph	875,713	5,105,039	21	2,746	855,802 (0.977)

sparks. To be concrete, using the library **`Par`** *Monad*, the decision of spawning new stages is done by the scheduler. Hence, dynamic pipelines hardly can be implemented using sparks.

Representation of stages. Stages are represented as *monadic computations*. These monadic computations are spawned in parallel through the use of the `async` combinator. Thus, different threads continue reading and writing from channels without blocking progress if data are available. Scripts in filter stages are also described as *monadic computations*. However, they are not spawned in different threads because, according to the DP computation model, they must run sequentially in the same filter instance (i.e., in the same thread).

Channels. Regarding channels, there are several techniques to communicate threads or sparks in Haskell like **`MVar`** or concurrent safe mechanisms like Software Transactional Memory (STM) [13]. At the same time, in the Haskell library ecosystem, there are `Channels` abstractions based on previously mentioned communication techniques. In that sense, for conducting the communication between dynamic stages and data flowing in a dynamic pipeline, we have selected the `unagi-chan` library[12] which provides the following advantages to our solution: Firstly, **`MVar`** channels without using STM reducing overhead. STM is not required in a dynamic pipeline because each specific stage running in a separated thread, can only access its `I/O` channels for reading/writing accordingly, and these operations are not concurrently shared by other threads (stages) for the same channels. Second, non-blocking channels. The `unagi-chan` library contains blocking and non-blocking channels for reading. This aspect is key to speeding up the implementation. Third, the library is optimized for $x86$ architectures using low-level `fetch-and-add` instructions. Finally, `unagi-chan` is $100x$ faster[13] on benchmarks compared to STM and baseline **`Chan`** implementations.

5. Empirical evaluation

The empirical study of the implementation of the algorithm presented in Section 3, DP_{WCC}, aims at evaluating the proposed Dynamic Pipeline framework $\text{DPF}_{\text{Haskell}}$. The current version of $\text{DPF}_{\text{Haskell}}$ is available as a Haskell package at *Hackage: The Haskell Package Repository*, https://hackage.haskell.org/package/dynamic-pipeline. For the experiments, we have implemented two solutions to the problem of weakly connected components:

- iDP_{WCC}: baseline (*ad hoc*) implementation of the weakly connected components as a dynamic pipeline.
- iDPF_{WCC}: implementation of the weakly connected components on top of the $\text{DPF}_{\text{Haskell}}$ framework.

Our goal is to answer the following research questions: **RQ1)** Does iDPF_{WCC} exhibit similar execution time performance compared with iDP_{WCC}? **RQ2)** Does iDPF_{WCC} enhance the continuous behavior with respect to the iDP_{WCC}? We have configured the following setup to assess our research questions.

5.1. Datasets

The experiments are performed on networks of the benchmark SNAP [24]. The selected networks correspond to complex undirected graphs of different sizes and connected components; Table 1 reports on the main characteristics of these undirected graphs.

5.2. Haskell setup

We use the following Haskell setup for the iDP_{WCC} implementation: GHC version 8.10.4, `bytestring 0.10.12.0`,[14] `containers 0.6.2.1`,[15] `relude 1.0.0.1`[16] and `unagi-chan 0.4.1.3`.[17] We disabled **`Prelude`** from the project with the language extension (language options) **`NoImplicitPrelude`**.[18] The decision of using the `relude` library instead of **`Prelude`** was made for

[12] https://hackage.haskell.org/package/unagi-chan.
[13] https://github.com/jberryman/unagi-chan.
[14] https://hackage.haskell.org/package/bytestring.
[15] https://hackage.haskell. org/package/containers.
[16] https://hackage.haskell.org/package/relude.
[17] https://github.com/jberryman/unagi-chan.
[18] https://downloads.haskell.org/ghc/8.8.4/docs/html/users_guide/.

Table 2
Total Execution times of each graph implemented with iDP_{WCC}. *MUT Time* is the time of actual code execution, while *GC Time* is the time the program spent doing garbage collection. *Total Execution time* is the sum of both times. For all these metrics lower is better. The parameters that characterize each graph affect the performance of iDP_{WCC} negatively, being Google Web Graph the one that consumes more *MUT Time* and *GC Time*.

Network	Exec Param	MUT Time	GC Time	Total Time
Enron Emails	`+RTS -N4 -s`	2.797s	0.942s	3.746s
Astro Physics Coll Net	`+RTS -N4 -s`	2.607s	1.392s	4.014s
Google Web Graph	`+RTS -N8 -s`	137.127s	218.913s	356.058s

the sake of safeness and freedom degrees in the development of the DPF. With the `relude` package, it is possible to decide what to include and avoid using some partial functions considered badly defined in **`Prelude`**. Regarding compilation flags (GHC options), we compiled with `-threaded`, `-O3`, `-rtsopts`, `-with-rtsopts=-N`. Since we are using `stack` version 2.5.1[19] as a build tool on top of GHC, the build command is `stack build`.[20] The setup for iDPF_{WCC} is the same, except that we use the `dynamic-pipeline 0.3.2.0`[21] library written as part of this work.

5.3. Experiments definition

E1: Implementation analysis. We evaluate the performance of the dynamic pipeline implementations for enumerating weakly connected components. We run the option +RTS -s flags to enable the GHC runtime statistics. For each implementation, we compute the metrics *MUT Time*, which is the time spent running the program itself, and *GC Time*, which is the time spent running the garbage collector. The *Total Execution Time* is the sum of both in seconds. At the same time, we check the correctness of the output by computing the number of WCC generated by the algorithm against the already known topology. This experiment provides evidence to answer the research question **RQ1**.

E2: Benchmark analysis. This experiment measures *Average Running Time*. The *Average Running Time* is the average running time of 1000 resamples using the `criterion` tool. In each sample, the running time is measured from the beginning of the program execution until the last answer is produced. This experiment will help to answer the research question **RQ1**.

E3: Continuous behavior. Therefore, it is important to evaluate the impact of implementing a dynamic pipeline on top of a DPF, in this case the $\text{DPF}_{\text{Haskell}}$. In this benchmark, we use `dief@t` [1], a relevant metric to measure diefficiency. The metric `dief@t` quantifies the amount of results generated during the first t time units of execution, relative to the results generated by the entire execution of the program. The higher the value of `dief@t`, the better the efficiency. The values of `dief@t` are calculated using the Diefficiency Metrics Tool, `diefpy`.[22] The `diefpy` tool generates radial plots reporting the comparison of `dief@t` and other non-continuous metrics: **i)** Completeness (Comp), which is the total number of answers produced. **ii)** Time for the first tuple (TFFT), which measures the elapsed time to produce the first answer. **iii)** Execution Time (ET) which measures the elapsed time to complete the execution of a query. **iv)** Throughput (T) which measures the total number of answers produced after evaluating a query divided by its execution time ET. Based on the reported values of `dief@t`, we aim at answering research question **RQ2**.

5.4. Discussion of observed results

Experiment: E1. Table 2 reports on the execution of iDP_{WCC} in the evaluated graphs. It is important to note that since the first two graphs are smaller in the number of edges compared to *Google Web Graph*, running them with 8 cores, as indicated by the `-N` parameters, does not affect the final speedup. GHC does not distribute threads to extra cores because it handles the load with 4 cores. Table 2 shows remarkable execution times for the first two graphs. This is not the case for the Google Web Graph. According to Table 1, the larger weakly connected components of all graphs have more than 90% of the vertices. However, this fact does not negatively affect the total execution time for the Enron Emails and Astro Physics Coll graphs, while it does affect the largest graph, the Google Web Graph. The results indicate the need for more sophisticated techniques to implement the DF_{WCC} algorithm. These techniques should avoid getting stuck while waiting to collect all vertices in filters to speed up execution. The number of connected components is the same in all cases.
Table 3 shows the *total execution time* for each of the networks implemented with iDPF_{WCC}. We observe similar execution time values compared to Table 2. In fact, for the Enron Emails and Astro Physics Coll graphs, all times are better than those obtained with iDP_{WCC}. The time for the Google Web graph is slightly worse. We have to keep in mind that $\text{DPF}_{\text{Haskell}}$ adds some overhead compared to just running the code. According to these results, we can partially answer **RQ1**, because the implementation of iDPF_{WCC} has similar performance compared to iDP_{WCC}.

[19] https://docs.haskellstack.org/en/stable/README/.
[20] For more information about package.yaml or cabal file, see https://github.com/jproyo/upc-miri-tfm/tree/main/connected-comp.
[21] https://hackage.haskell.org/package/dynamic-pipeline.
[22] https://github.com/SDM-TIB/diefpy/.

Table 3
Total Execution times of each graph implemented with $\mathrm{iDPF}_{\mathrm{WCC}}$. *MUT Time* is the time of actual code execution, while *GC Time* is the time the program spent doing garbage collection. *Total Execution time* is the sum of both times. For all these metrics lower is better. The parameters that characterize each graph affect the performance of $\mathrm{iDPF}_{\mathrm{WCC}}$ negatively, being Google Web Graph the one that consumes more *MUT Time* and *GC Time*.

Network	Exec Param	MUT Time	GC Time	Total Time
Enron Emails	`+RTS -N4 -s`	1.795s	0.505s	2.314s
Astro Physics Coll Net	`+RTS -N4 -s`	2.294s	1.003s	3.311s
Google Web Graph	`+RTS -N8 -s`	169.381s	270.784s	440.176s

Table 4
Comparison of execution times between $\mathrm{iDPF}_{\mathrm{WCC}}$ vs. $\mathrm{iDP}_{\mathrm{WCC}}$. The graph with the higher speed-up is highlighted in **bold**. The parameters that characterize each graph impact on the performance of $\mathrm{iDPF}_{\mathrm{WCC}}$ and $\mathrm{iDP}_{\mathrm{WCC}}$. Google Web Graph is the most complex graph and its characteristics affect the performance of $\mathrm{iDPF}_{\mathrm{WCC}}$.

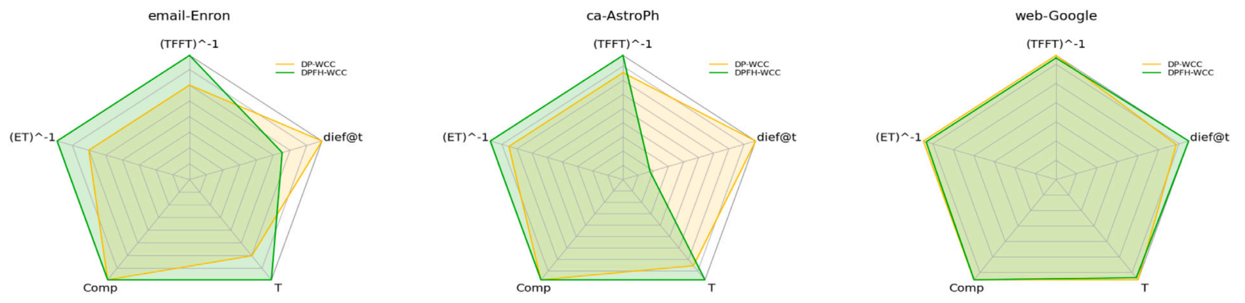
Network	$\mathrm{iDPF}_{\mathrm{WCC}}$	$\mathrm{iDP}_{\mathrm{WCC}}$	Speed-up
Enron Emails	4.30s	4.68s	0.91
Astro Physics Coll Net	4.76s	4.98s	**0.95**
Google Web Graph	456s	386s	-1.18

Table 5
Continuous behavior of $\mathrm{iDP}_{\mathrm{WCC}}$ and $\mathrm{iDPF}_{\mathrm{WCC}}$. The metric `dief@t` quantifies the continuous generation of weakly connected components; `dief@t` is a higher-is-better metric. All approaches continuously enumerate the weakly connected components. However, the characteristics of the graphs (i.e., the size, density, and number of weakly connected components) affect their behavior. For example, $\mathrm{iDP}_{\mathrm{WCC}}$ enumerates the results more continuously in Enron Emails and Astro Physics Coll, where the number of weakly connected components is relatively small. On the other hand, Google Web Graph has a large number of weakly connected components, and one of them is very large. As a result, the $\mathrm{iDP}_{\mathrm{WCC}}$ encounters obstacles in generating the largest element, which subsequently undermines its sustained performance. This correlation is consistent with the conclusions of Navarro et al. [30], who state that imbalances in block distribution lead to inefficiencies in stage execution. To address this challenge, it is imperative to develop optimization techniques that prioritize fair workload and data distribution, thereby promoting a more efficient overall process.

Graph	Implementation	dief@t Metric
Enron Emails	$\mathrm{iDPF}_{\mathrm{WCC}}$	1.38×10^6
	$\mathrm{iDP}_{\mathrm{WCC}}$	1.98×10^6
Astro Physics Coll	$\mathrm{iDPF}_{\mathrm{WCC}}$	1.80×10^5
	$\mathrm{iDP}_{\mathrm{WCC}}$	8.77×10^5
Google Web Graph	$\mathrm{iDPF}_{\mathrm{WCC}}$	1.29×10^7
	$\mathrm{iDP}_{\mathrm{WCC}}$	1.17×10^7

Experiment: E2. With the results of the experiment *E1* we have a partial answer to **RQ1**. In fact, we found that, depending on the implemented algorithm, the distribution of the input data can affect the performance and the parallelization of a dynamic pipeline. Table 4 summarizes the comparison of the *average execution times* of $\mathrm{iDP}_{\mathrm{WCC}}$ and $\mathrm{iDPF}_{\mathrm{WCC}}$. In both implementations, the graph that performs slightly worse is Google Web, but the difference is not significant enough considering the overhead introduced by $\mathrm{DPF}_{\mathrm{Haskell}}$. The results of the experiments *E1* and *E2* allow to answer the search question **RQ1** and confirm that $\mathrm{iDPF}_{\mathrm{WCC}}$ has a similar performance compared to the *ad hoc* implementation $\mathrm{iDP}_{\mathrm{WCC}}$. This result is really encouraging for the further development of a Dynamic Pipeline framework.

Experiment: E3. In the case of Enron Emails and Astro Physics Coll, $\mathrm{iDP}_{\mathrm{WCC}}$ continuously computes and enumerates results because the size and density of the components are uniform. As a result, the values of `dief@t` are higher in $\mathrm{iDP}_{\mathrm{WCC}}$ than in $\mathrm{iDPF}_{\mathrm{WCC}}$, even though $\mathrm{iDPF}_{\mathrm{WCC}}$ continuously produces results. Table 5 reports values `dief@t` that quantify these observations. Complementary radar plots in Fig. 5 show the values of the metrics Completeness (Comp), Time for the first tuple (TFFT), Execution Time (ET), and Throughput (T); for all the metrics shown higher is better; and $\mathrm{iDP}_{\mathrm{WCC}}$ is shown in yellow, while $\mathrm{iDPF}_{\mathrm{WCC}}$ is shown in green. As shown, all approaches produce all results, i.e. the completeness is the same for $\mathrm{iDPF}_{\mathrm{WCC}}$ and $\mathrm{iDP}_{\mathrm{WCC}}$ in the three graphs. $\mathrm{iDP}_{\mathrm{WCC}}$ produces more continuous results in Enron Emails and Astro Physics Coll. However, $\mathrm{iDPF}_{\mathrm{WCC}}$ produces the first results faster (i.e., the inverse of the time for the first tuple (TFFT^{-1}) is better) and has a better elapsed time (i.e., the inverse of the execution time ET^{-1}). This behavior may be caused by the fact that both graphs contain a more balanced number of complex components, and there are no



(a) email-Enron Diefficency Metrics iDPF$_{\mathrm{WCC}}$ vs. iDP$_{\mathrm{WCC}}$ Radar Plot

(b) ca-AstroPh Diefficency Metrics iDPF$_{\mathrm{WCC}}$ vs. iDP$_{\mathrm{WCC}}$ Radar Plot

(c) web-Google Diefficency Metrics iDPF$_{\mathrm{WCC}}$ vs. iDP$_{\mathrm{WCC}}$ Radar Plot

Fig. 5. Continuous behavior of iDP$_{\mathrm{WCC}}$ and iDPF$_{\mathrm{WCC}}$. Radial plots report on T, TFFT, `dief@t`, ET and Comp. Consistent with the results in Table 5, all approaches continuously enumerate the weakly connected components. However, the number and shape of the weakly connected components in the Google Web Graph prevent a more continuous enumeration of results because iDP$_{\mathrm{WCC}}$ gets stuck generating the larger component. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

components where iDPF$_{\mathrm{WCC}}$ or iDP$_{\mathrm{WCC}}$ gets stuck. On the contrary, the observed results suggest a different story in the Google Web Graph, where both approaches perform almost the same in the three graphs. As discussed earlier, the Google Web Graph contains a large number of weakly connected components, and one of them is very large. iDPF$_{\mathrm{WCC}}$ seems to be slightly more continuous than iDP$_{\mathrm{WCC}}$, according to its `dief@t` values. In conclusion, regarding **RQ2**, although iDP$_{\mathrm{WCC}}$ is more continuous in Enron Emails and Astro Physics Coll, iDPF$_{\mathrm{WCC}}$ is faster on those and more continuous in Google Web Graph. Thus, these results suggest that iDPF$_{\mathrm{WCC}}$ preserves the continuity approach, and in some cases it may even be faster than iDP$_{\mathrm{WCC}}$.

6. Related work

Streaming Processing Techniques. The development of streaming processing techniques has stimulated areas of massive data processing for data mining algorithms, big data analytics, IoT applications, etc. *Data streaming* has been studied using different approaches ([7] and see [33] for a survey) that allow us to efficiently process a large amount of data with an intensive level of parallelization. There are two main different models of stream parallelization: *data parallelism* and *pipeline parallelism*. [11,22]. According to the data parallelism approach, data is divided into non-overlapping subsets that are processed in parallel, so that all computations performed in parallel on different subsets of data are independent. One of the main challenges when using this approach is to properly tune the workload a priori. A common model that has been successful over the last decade is MapReduce (MR) [6]. Various frameworks or tools such as Hadoop,[23] Spark,[24] etc. efficiently support this computational model. One of the main advantages of this type of model is the ability to implement stateless operators [33].
Data is processed in different threads or processors without contextual information. However, when it is necessary to be aware of the context, parallelization is penalized and each computational step should be fully computed before proceeding with the others (*blocking*), i.e. the `reduce` operation on many frameworks or tools. Thus, the data parallelism approach is hardly viable for delivering results incrementally. Flink[25] is a stream processing framework based on stateful operators. It is important to note that it is necessary to use the *Stateful Functions* library to implement elastic stateful stream processing application using this framework. However, to our knowledge, the documentation of this library is still not sufficient, and in addition, developers have to deal with many problems if the ingestion of data is not done by *Kafka*.[26] Furthermore, due to its computation model [16], it is not natural to deliver data continuously in Flink applications.
In contrast to these frameworks mentioned above, the Dynamic Pipeline approach naturally exploits *elastic pipeline parallelism* and *stateful operators* (filter instances stages). No matter what the data source is, Source is set up by the user (this feature is supported by the IDE in the DPF). Therefore, the splitting and grouping of (possibly overlapping) data is done as needed. That is, the workload is self-adapted by the dynamic pipelines. In addition, each stage produces results as soon as they are ready, allowing continuous generation of responses. This approach allows to handle, in a very natural way, real-time stream processing and clustering without knowing the number of clusters beforehand.[27]
Strategies for Dynamic Pipelines. Creating groups of input data and applying a function to each group, as we do when using the Dynamic Pipeline approach, reminds us of the divide and conquer pattern for solving problems [35]. However, on the one hand, divide and conquer is useful for solving problems where data is limited and *in-memory*. Instead, our approach is oriented towards solving problems where data is not necessarily bounded and can be ingested dynamically, i.e. data streams. Moreover, the results

[23] https://hadoop.apache.org/.
[24] https://spark.apache.org/.
[25] https://flink.apache.org/.
[26] https://kafka.apache.org/.
[27] It is well known that one of the main concerns in the *clustering problem* is the estimation of the number of clusters.

are output as they are generated, i.e., results are generated incrementally. In particular, according to stream processing techniques, results are generated using a *window-based* approach. On the contrary, in the divide and conquer paradigm, the problem remains divided into subproblems, in terms of the size of the problem instead of a grouping relation, until the ones that are simple enough to start producing solutions are reached. In the case of the DP approach, the input data is grouped into (multi)subsets according to a grouping property, a specific function is applied to these subsets, and the results of these applications are combined to produce results. Regarding the MapReduce programming model [18], it is a *data parallel* processing model. Map and Reduce functions are blocking operations [23]. This means that in a MapReduce workflow solution all tasks must be completed to move forward to the next stage. This fact causes performance degradation and makes it difficult to support continuous real-time processing.

Metrics in Streaming Processing Techniques. The primary metrics considered when evaluating streaming processing pipelines with massive input data are *latency* and *throughput* and resource utilization [38]. Latency measures how long it takes the framework to deliver a result. Throughput measures the amount of data processed in a unit of time. A good pipeline framework should have low latency and high throughput. However, as mentioned earlier, there are stream processing problems where incremental result generation is critical. The time it takes to generate all the results can affect the continuous behavior despite latency and throughput. This issue can affect the solution of problems in domains such as biomedicine or social sciences, where answers must be produced in real time. In fact, there are domains where many applications need to process large amounts of data, where users do not need all the results, but only a fraction of them to start making decisions and performing analysis. In addition, users of these types of applications are able to manage the resources they can afford. Thus, the more resources users can afford, the more data or faster results they will receive. This method of delivering results incrementally in resource-intensive processes, is known as the *pay-as-you-go model* [31]. As a result, measuring only the latency and throughput of stream processing frameworks does not illustrate the suitability of a framework when continuous performance is required. The Dynamic Pipeline framework provides the basis for implementing a robust tool based on the Dynamic Pipeline approach. As the empirical evaluation results suggest, it can efficiently generate responses continuously.

7. Conclusions and future work

The growing demand for streaming processing has spurred the development of many computational models for handling this type of processing. One of these computational models is the Dynamic Pipeline approach. This work defines a Dynamic Pipeline framework and establishes the implementation requirements while outlining the targeted problems that these frameworks aim to address. As a proof of concept, the paper provides insight into the essential components of implementing a Dynamic Pipeline framework using Haskell. This exploration takes into account the relevant features, including versions and libraries, that contribute to the efficient realization of a Dynamic Pipeline framework.

In addition, this paper exemplifies the practical application of the provided Dynamic Pipeline framework by demonstrating its effectiveness in solving the challenge of identifying weakly connected graph components. To validate the effectiveness of the solutions, empirical evaluations are performed on a contemporary benchmark, the Stanford Network Analysis Platform (SNAP) [24]. This benchmark consists of three different graphs of varying complexity, which pose rigorous challenges to the evaluated implementations (namely $\mathsf{iDPF}_{\mathsf{WCC}}$ and $\mathsf{iDP}_{\mathsf{WCC}}$) of Dynamic Pipeline framework.

The paper also illustrates the category of problems that can be solved using the DP approach, along with an elucidation of its computational framework. In particular, the DP approach facilitates the creation of concise implementations, shielding the user from the complexities of concurrency and parallelism management. In addition, the computational model allows for the rapid generation of results, providing results as soon as they are generated. Finally, while not necessarily indicative of superior programming quality, code developed using the DP approach is significantly more compact than the *ad hoc* programs created within the same programming language that accommodates the Dynamic Pipeline approach.

While not exhaustive in scope, the empirical evaluation of the two implementations designed to compute weakly connected graph components effectively underscores their suitability and robustness in realizing a Dynamic Pipeline framework within the Haskell programming environment. The use of `dief@t` measurements provides compelling evidence for the consistent behavior of both $\mathsf{iDPF}_{\mathsf{WCC}}$ and $\mathsf{iDP}_{\mathsf{WCC}}$. Furthermore, the results presented shed light on the key parameters of these implementations and demonstrate their efficient performance. These results not only confirm the feasibility of implementing a DP in Haskell, but also highlight the language's ability to handle the demands of the WCC problem without compromising execution speed or memory allocation. We anticipate that these results will inspire the community to venture into implementing other algorithms in Dynamic Pipeline framework, thereby unlocking its potential benefits in diverse problem domains.

Based on the insights gained from the evaluation of the Dynamic Pipeline framework, our next endeavors involve the development of an improved iteration, as advocated by [27], aimed at achieving higher levels of parallelization through the use of sparks within the **`Par`** *Monad*. This strategic shift is expected to push the framework towards greater scalability. Another facet of our future plan is to apply the Dynamic Pipeline approach to various subgraph enumeration problems, such as bi-triangles or graphlets, which are indispensable components of efficient network analysis. Furthermore, we intend to broaden the scope of the applicability of the Dynamic Pipeline approach by implementing versions in both the GO[28] and Elixir[29] programming languages. This pursuit is inspired by the recognition that these languages adeptly address scalability concerns in complex scenarios. The encouraging results reported

[28] https://go.dev/.

[29] https://elixir-lang.org/.

by Zoltan et al. [32] in the context of the GO implementation serve as a promising foundation. We are committed to developing approaches that are seamlessly consistent with the principles outlined in Section 4. Through these upcoming research initiatives, we intend to continually provide the programming community with frameworks that effectively address the challenges posed by the escalating demand for streaming processing solutions. In addition, we intend to formulate optimization methods tailored to adapt the scheduling of a dynamic pipeline to the characteristics of the input data. These adaptive execution strategies aim to ensure a harmoniously balanced workload distribution across the various stages of the pipeline.

CRediT authorship contribution statement

Edelmira Pasarella: Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Conceptualization. **Maria-Esther Vidal:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Conceptualization. **Cristina Zoltan:** Supervision, Conceptualization. **Juan Pablo Royo Sales:** Writing – original draft, Software, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://github.com/jproyo/dynamic-pipeline

Acknowledgements

This work has been partially supported by MCIN/ AEI /10.13039/501100011033 under grant PID2020-112581GB-C21 and by the Leibniz Association Germany, “Leibniz Best Minds: Programme for Women Professors”, project TrustKG-Transforming Data in Trustable Insights; Grant P99/2020. The authors would like to express the sincere gratitude to the anonymous reviewers for their insightful comments and constructive feedback, which greatly contributed to the improvement of this manuscript. Their expertise and dedication have been invaluable in shaping the final version of the paper.

References

[1] M. Acosta, M.-E. Vidal, Y. Sure-Vetter, Diefficiency metrics: measuring the continuous efficiency of query processing approaches, in: International Semantic Web Conference, Springer, 2017, pp. 3–19.
[2] M. Ahmed, R. Seraj, S.M.S. Islam, The k-means algorithm: a comprehensive survey and performance evaluation, Electronics 9 (8) (2020) 1295.
[3] E.A. Bender, Partitions of multisets, Discrete Math. 9 (4) (1974) 301–311.
[4] W.D. Blizard, The development of multiset theory, Mod. Log. 1 (4) (1991) 319–352.
[5] J. Buisson, F. Andre, J.-L. Pazat, Performance and practicability of dynamic adaptation for parallel computing, in: 2006 15th IEEE International Conference on High Performance Distributed Computing, 2006, pp. 331–332.
[6] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[7] T. Dunning, E. Friedman, Streaming Architecture: New Designs Using Apache Kafka and MapR Streams, O'Reilly Media, Inc., 2016.
[8] M. Fowler, Domain-Specific Languages, Pearson Education, 2010.
[9] M.-d.-M. Gallardo, L. Panizo, Trace analysis using an event-driven interval temporal logic, in: International Symposium on Logic-Based Program Synthesis and Transformation, Springer, 2019, pp. 177–192.
[10] B. Gedik, Generic windowing support for extensible stream processing systems, Softw. Pract. Exp. 44 (9) (2014) 1105–1128.
[11] M.I. Gordon, W. Thies, S. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, ACM SIGOPS Oper. Syst. Rev. 40 (5) (2006) 151–162.
[12] J.L. Gross, J. Yellen, Handbook of Graph Theory, CRC Press, 2003.
[13] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2005, pp. 48–60.
[14] J. Herath, Y. Yamaguchi, N. Saito, T. Yuba, Dataflow computing models, languages, and machines for intelligence computations, IEEE Trans. Softw. Eng. 14 (12) (1988) 1805–1828.
[15] W.A. Howard, The formulae-as-types notion of construction, in: J.R. Hindley, J.P. Seldin (Eds.), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980.
[16] F. Hueske, V. Kalavri, Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications, O'Reilly Media, 2019.
[17] S. Im, B. Moseley, X. Sun, Efficient massively parallel methods for dynamic programming, in: H. Hatami, P. McKenzie, V. King (Eds.), Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017, ACM, 2017, pp. 798–811.
[18] H. Karloff, S. Suri, S. Vassilvitskii, A Model of Computation for MapReduce, in: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, 2010, pp. 938–948, https://epubs.siam.org/doi/abs/10.1137/1.9781611973075.76.
[19] S.N. Khezr, N.J. Navimipour, Mapreduce and its applications, challenges, and architecture: a comprehensive review and directions for future research, J. Grid Comput. 15 (2017) 295–321.
[20] B. Le Goff, P. Le Guernic, J. Araoz Durand, Semi-granules and schielding for off-line scheduling, Rapports de recherche- INRIA.
[21] H.F. Ledgard, M. Marcotty, A genealogy of control structures, Commun. ACM 18 (11) (1975) 629–639.
[22] I.-T.A. Lee, C.E. Leiserson, T.B. Schardl, Z. Zhang, J. Sukha, On-the-fly pipeline parallelism, ACM Trans. Parallel Comput. 2 (3) (2015) 1–42.
[23] K.-H. Lee, Y.-J. Lee, H. Choi, Y.D. Chung, B. Moon, Parallel data processing with mapreduce: a survey, ACM SIGMOD Rec. 40 (4) (2012) 11–20.
[24] J. Leskovec, A. Krevl, SNAP datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, June 2014.

[25] J. Li, D. Maier, K. Tufte, V. Papadimos, P.A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005, pp. 311–322.
[26] S. Marlow Haskell, Async library, https://hackage.haskell.org/package/async. (Accessed 17 April 2021).
[27] S. Marlow, Parallel and concurrent programming in Haskell, in: V. Zsók, Z. Horváth, R. Plasmeijer (Eds.), CEFP 2011, in: LNCS, vol. 7241, O'Reilly Media, Inc., 2012, pp. 339–401.
[28] S. Marlow, P. Maier, H.-W. Loidl, M.K. Aswad, P. Trinder, Seq no more: better strategies for parallel Haskell, ACM SIGPLAN Not. 45 (11) (2010) 91–102.
[29] S. Marlow, R. Newton, S. Peyton Jones, A monad for deterministic parallelism, ACM SIGPLAN Not. 46 (12) (2011) 71–82.
[30] A. Navarro, R. Asenjo, S. Tabik, C. Cascaval, Analytical modeling of pipeline parallelism, in: 2009 18th International Conference on Parallel Architectures and Compilation Techniques, IEEE, 2009, pp. 281–290.
[31] T.T. Nguyen, M. Weidlich, H. Yin, B. Zheng, Q.H. Nguyen, Q.V.H. Nguyen Factcatch, Incremental pay-as-you-go fact checking with minimal user effort, in: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, 2020, pp. 2165–2168.
[32] E. Pasarella, M.-E. Vidal, C. Zoltan, Comparing MapReduce and pipeline implementations for counting triangles, Electron. Proc. Theor. Comput. Sci. 237 (2017) 20–33.
[33] H. Röger, R. Mayer, A comprehensive survey on parallelization and elasticity in stream processing, ACM Comput. Surv. 52 (2) (2019) 1–37.
[34] J.P. Royo-Sales, E. Pasarella, C. Zoltan, M.-E. Vidal, Towards a dynamic pipeline framework implemented in (parallel) Haskell, in: PROLE2021, SISTEDES, 2021, http://hdl.handle.net/11705/PROLE/2021/017.
[35] D.R. Smith, The design of divide and conquer algorithms, Sci. Comput. Program. 5 (1985) 37–58.
[36] A.S. Tanenbaum, T. Austin, Structured Computer Organization, Pearson, ISBN 978-0132916523, 2013.
[37] N. Ukey, Z. Yang, W. Yang, B. Li, R. Li, knn join for dynamic high-dimensional data: a parallel approach, in: Z. Bao, R. Borovica-Gajic, R. Qiu, F. Choudhury, Z. Yang (Eds.), Databases Theory and Applications, Springer Nature Switzerland, Cham, 2024, pp. 3–16.
[38] G. Van Dongen, D. Van den Poel, Evaluation of stream processing frameworks, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1845–1858.
[39] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, Y. Etsion, Hybrid dataflow/von-neumann architectures, IEEE Trans. Parallel Distrib. Syst. 6 (25) (2014) 1489–1509.
[40] C. Zoltan, E. Pasarella, J. Araoz, M.-E. Vidal, The dynamic pipeline paradigm, in: PROLE2019, SISTEDES, 2019, http://hdl.handle.net/11705/PROLE/2019/017.