A Formal Semantics for Data Analytics Pipelines

Technical Report

Maurizio Drocco, Claudia Misale, Guy Tremblay, Marco Aldinucci

May 5, 2017

Abstract

In this report, we present a new programming model based on Pipelines and Operators, which are the building blocks of programs written in PiCo, a DSL for Data Analytics Pipelines. In the model we propose, we use the term Pipeline to denote a workflow that processes data collections—rather than a computational process—as is common in the data processing community.

The novelty with respect to other frameworks is that all PiCo operators are polymorphic with respect to data types. This makes it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context, i.e., the previous and following stages in the pipeline. Notice that in other mainstream frameworks, such as Spark, the update of a pipeline by changing a transformation with another is not necessarily trivial, since it may require the development of an input and output proxy to adapt the new transformation for the calling context.

In the same line, we provide a formal framework (i.e., typing and semantics) that characterizes programs from the perspective of how they transform the data structures they process—rather than the computational processes they represent. This approach allows to reason about programs at an abstract level, without taking into account any aspect from the underlying execution model or implementation.

1 Introduction

Big Data is becoming one of the most (ab)used buzzword of our times. In companies, industries, academia, the interest is dramatically increasing and everyone wants to "do Big Data", even though its definition or role in analytics is not completely clear. From a high-level perspective, Big Data is about extracting knowledge from both structured and unstructured data. This is a useful process for big companies such as banks, insurance, telecommunication, public institutions, and so on, as well as for business in general. Extracting knowledge from Big Data requires tools satisfying strong requirements with respect to programmability — that is, allowing to easily write programs and algorithms to analyze data — and performance, ensuring scalability when running analysis and queries on multicore or cluster of multicore nodes. Furthermore, they need to cope with input data in different formats, e.g. batch from data marts, live stream from the Internet or very high-frequency sources. In the last decade, a large number of frameworks for Big Data processing has been implemented addressing these issues.

Their common aim is to ensure ease of programming by providing a unique framework addressing both batch and stream processing. Even if they accomplish this task, they often lack of a clear semantics of their programming and execution model. For instance, users can be provided with two different data models for representing collections and streams, both supporting the same operations but often having different semantics.

We advocate a new Domain Specific Language (DSL), called **Pi**peline **Co**mposition (PiCo), designed over the presented layered Dataflow conceptual framework [9]. PiCo programming model aims at *easing the programming* of Analytics applications by two design routes: 1) unifying data access model, and 2) decoupling processing from data layout.

Both design routes undertake the same goal, which is the raising of the level of abstraction in the programming and the execution model with respect to mainstream approaches in tools (Spark [11], Storm [10], Flink [2] and Google Dataflow [1]) for Big Data analytics, which typically force the specialization of the algorithm to match the data access and layout. Specifically, data transformation functions (called *operators* in PiCo) exhibit a different functional types when accessing data in different ways.

For this reason, the source code should be revised when switching from one data model to the next. This happens in all the above mentioned frameworks and also in the abstract Big Data architectures, such as the Lambda [6] and Kappa architectures [5]. Some of them, such as the Spark framework, provide the runtime with a module to convert streams into micro-batches (Spark Streaming, a library running on Spark core), but still a different code should be written at user-level. The Kappa architecture advocates the opposite approach, i.e., to "streamize" batch processing, but the streamizing proxy has to be coded. The Lambda architecture requires the implementation of both a batch-oriented and a stream-oriented algorithm, which means coding and maintaining two codebases per algorithm.

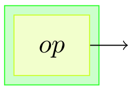
PiCo fully decouples algorithm design from data model and layout. Code is designed in a fully functional style by composing stateless *operators* (i.e., transformations in Spark terminology). As we discuss in this report, all PiCo operators are polymorphic with respect to data types. This makes it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context, i.e., the previous and following stages in the pipeline. Notice that in other mainstream frameworks, such as Spark, the update of a pipeline by changing a transformation with another is not necessarily trivial, since it may require the development of an input and output proxy to adapt the new transformation for the calling context.

This report proceeds as follows. We formally define the syntax of a program, which is based on Pipelines and operators whereas it hides the data structures produced and generated by the program. Then we provide the formalization of a minimal type system defining legal compositions of operators into Pipelines. Finally, we provide a semantic interpretation that maps any PiCo program to a functional Dataflow graph, representing the transformation flow followed by the processed collections.

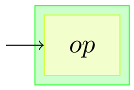
2 Syntax

We propose a programming model for processing data collections, based on the Dataflow model. The building blocks of a PiCo program are *Pipelines* and *Operators*, which we investigate in this section. Conversely, *Collections* are not included in the syntax and they are introduced in Section 3.1 since they contribute at defining the type system and the semantic interpretation of PiCo programs.

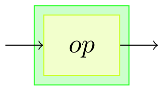
2.1 Pipelines



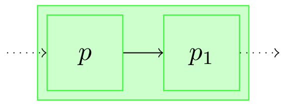
(a) Source



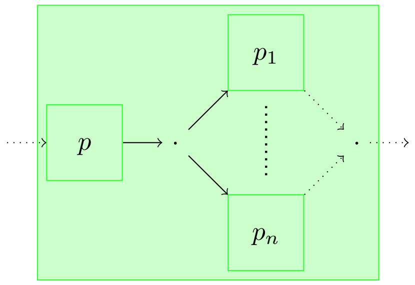
(b) Sink



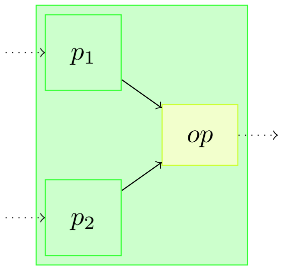
(c) Processing



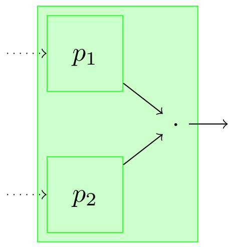
(d) Linear TO



(e) Non-linear TO



(f) PAIR



(g) MERGE

Figure 1: Graphical representation of PiCo Pipelines

The cornerstone concept in the Programming Model is the *Pipeline*, basically a DAG-composition of processing *operators*. Pipelines are built according to the following grammar[1]:

⟨*Pipeline*⟩ ::= NEW ⟨`unary`-*operator*⟩
  | TO ⟨*Pipeline*⟩ ⟨*Pipeline*⟩ … ⟨*Pipeline*⟩
  | PAIR ⟨*Pipeline*⟩ ⟨*Pipeline*⟩ ⟨`binary`-*operator*⟩
  | MERGE ⟨*Pipeline*⟩ ⟨*Pipeline*⟩

We categorize Pipelines according to the number of collections they take as input and output:

- A source Pipeline takes no input and produces one output collection

[1] For simplicity, here we introduce the non-terminal `unary`-*operator* (resp. `binary`-*operator*) that includes core and partitioning unary (resp. binary) operators.

Pipeline	Structural properties	Behavior
NEW *op*	-	data is processed by operator *op* (i.e., *unary* Pipeline)
TO $p\ p_1\ \dots\ p_n$	associativity for linear Pipelines: TO (TO $p_A\ p_B$) $p_C \equiv$ TO p_A (TO $p_B\ p_C$) $\equiv$ $p_A \mid p_B \mid p_C$ destination commutativity: TO $p\ p_1 \dots p_n \equiv$ TO $p\ p_{\pi(1)} \dots p_{\pi(n)}$ for any π permutation of $1..n$	data from Pipeline p is sent to all Pipelines p_i (i.e., broadcast)
PAIR $p_1\ p_2$ *op*	-	data from Pipelines p_1 and p_2 are pair-wise processed by operator *op*
MERGE $p_1\ p_2$	associativity: MERGE (MERGE $p_1\ p_2$) $p_3 \equiv$ MERGE p_1 (MERGE $p_2\ p_3$) $\equiv$ $p_1 + p_2 + p_3$ commutativity: MERGE $p_1\ p_2 \equiv$ MERGE $p_2\ p_1$	data from Pipelines p_1 and p_2 are merged, respecting the ordering in case of ordered collections

Table 1: Pipelines

- A sink Pipeline consumes one input collection and produces no output
- A processing Pipeline consumes one input collection and produces one output collection

A pictorial representation of Pipelines is reported in Figure 1. We refer to Figs. 1a, 1b and 1c as *unary* Pipelines, since they are composed by a single operator. Figs. 1e and 1d represent, respectively, linear (i.e., one-to-one) and branching (i.e., one-to-n) TO composition. Figs. 1f and 1g represent composition of Pipelines by, respectively, pairing and merging. A dotted line means the respective path may be void (e.g., a source Pipeline has void input path). Moreover, as we show in Section 3, Pipelines are not allowed to consume more than one input collection, thus both PAIR and MERGE Pipelines must have at least one void input path.

The meaning of each Pipeline is summarized in Table 1.

2.2 Operators

Operators are the building blocks composing a Pipeline. They are categorized according to the following grammar of core operator families:

$\langle$*core-operator*$\rangle$ ::= $\langle$*core-unary-operator*$\rangle$ | $\langle$*core-binary-operator*$\rangle$

$\langle$*core-unary-operator*$\rangle$ ::= $\langle$*map* $\rangle$ | $\langle$*combine*$\rangle$ | $\langle$*emit*$\rangle$ | $\langle$*collect*$\rangle$

$\langle$*core-binary-operator*$\rangle$ ::= $\langle$*b-map*$\rangle$ | $\langle$*b-combine*$\rangle$

The intuitive meanings of the core operators are summarized in Table 2.

Operator family	Categorization	Decomposition	Behavior
`map`	unary, element-wise	no	applies a user function to each element in the input collection
`combine`	unary, collective	yes	synthesizes all the elements in the input collection into an atomic value, according to a user-defined policy
`b-map`	binary, pair-wise	yes	the binary counterpart of `map`: applies a (binary) user function to each pair generated by pairing (i.e. zipping/joining) two input collections
`b-combine`	binary, collective	yes	the binary counterpart of `combine`: synthesizes all pairs generated by pairing (i.e. zipping/joining) two input collections
`emit`	produce-only	no	reads data from a source, e.g., regular collection, text file, tweet feed, etc.
`collect`	consume-only	no	writes data to some destination, e.g., regular collection, text file, screen, etc.

Table 2: Core operator families.

In addition to core operators, generalized operators can decompose their input collections by:

- partitioning the input collection according to a user-defined grouping policy (e.g., group by key)
- windowing the *ordered* input collection according to a user-defined windowing policy (e.g., sliding windows)

The complete grammar of operators follows:

$\langle$*operator*$\rangle$ ::= $\langle$*core-operator*$\rangle$
 | $\langle$*w-operator*$\rangle$ | $\langle$*p-operator*$\rangle$ | $\langle$*w-p-operator*$\rangle$

where `w-` and `p-` denote decomposition by windowing and partitioning, respectively.

For those operators *op* not supporting decomposition (cf. Table 2), the following structural equivalence holds: $op \equiv \texttt{w-}op \equiv \texttt{p-}op \equiv \texttt{w-p-}op$.

2.2.1 Data-Parallel Operators

Operators in the `map` family are defined according to the following grammar:

$\langle map \rangle ::= \texttt{map}\ f \mid \texttt{flatmap}\ f$

where f is a user-defined function (i.e., the *kernel* function) from a host language.[2] The former produces exactly one output element from each input element (one-to-one user function), whereas the latter produces a (possibly empty) bounded sequence of output elements for each input element (one-to-many user function) and the output collection is the merging of the output sequences.

Operators in the `combine` family synthesize all the elements from an input collection into a single value, according to a user-defined kernel. They are defined according to the following grammar:

$\langle combine \rangle ::= \texttt{reduce} \oplus \mid \texttt{fold+reduce}\ \oplus_1\ z \oplus_2$

The former corresponds to the classical reduction, whereas the latter is a two-phase aggregation that consists in the reduction of partial accumulative states (i.e., partitioned folding with explicit initial value). The parameters for the `fold+reduce` operator specify the initial value for each partial accumulator ($z \in S$, the initial value for the folding), how each input item affects the aggregative state ($\oplus_1 : S \times T \to S$, the folding function) and how aggregative states are combined into a final accumulator ($\oplus_2 : S \times S \to S$, the reduce function).

2.2.2 Pairing

Operators in the `b-map` family are intended to be the binary counterparts of `map` operators:

$\langle b\text{-}map \rangle ::= \texttt{zip-map}\ f \mid \texttt{join-map}\ f$
$\quad \mid\ \texttt{zip-flatmap}\ f \mid \texttt{join-flatmap}\ f$

The binary user function f takes as input pairs of elements, one from each of the input collections. Variants `zip-` and `join-` corresponds to the following pairing policies, respectively:

- zipping of ordered collections produces the pairs of elements with the same position within the order of respective collections
- joining of bounded collections produces the Cartesian product of the input collections

[2]Note that we treat kernels as terminal symbols, thus we do not define the language in which kernel functions are defined; we rather denote this aspect to a specific implementation of the model.

Analogously, operators in the `b-combine` family are the binary counterparts of `combine` operators.

2.2.3 Sources and Sinks

Operators in the `emit` and `collect` families model data collection sources and sinks, respectively:

$\langle \mathit{emit} \rangle$ ::= `from-file` file | `from-socket` socket | ...

$\langle \mathit{collect} \rangle$ ::= `to-file` file | `to-socket` socket | ...

2.2.4 Windowing

Windowing is a well-known approach for overcoming the difficulties stemming from the unbounded nature of stream processing. The basic idea is to process parts of some recent stream history upon the arrival of new stream items, rather than store and process the whole stream each time.

A windowing operator takes an ordered collection, produces a collection (with the same structure type as the input one) of windows (i.e., lists), and applies the subsequent operation to each window. Windowing operators are defined according to the following grammar, where ω is the windowing policy:

$\langle \mathit{w\text{-}operator} \rangle$ ::= `w-`$\langle \mathit{core\text{-}operator} \rangle$ ω

Among the various definitions from the literature, for the sake of simplicity we only consider policies producing *sliding windows*, characterized by two parameters, namely, a window size $|W|$—specifying which elements fall into a window—and a sliding factor δ—specifying how the window slides over the stream items. Both parameters can be expressed either in time units (i.e., time-based windowing) or in number of items (i.e., count-based windowing). In this setting, a windowing policy ω is a term $(|W|, \delta, b)$ where b is either `time` or `count`. A typical case is when $|W| = \delta$, referred as a *tumbling* policy.

The meaning of the supported windowing policies will be detailed in semantic terms (Section 4.1). Although the PiCo syntax only supports a limited class of windowing policies, the semantics we provide is general enough to express other policies such as session windows [4].

As we will show in Section 3, we rely on tumbling windowing to extend bounded operators[3] and have them deal with unbounded collections; for instance, `combine` operators are bounded and require windowing to extend them to unbounded collections.

[3]We say an operator is *bounded* if it can only deal with bounded collections.

2.2.5 Partitioning

Logically, partitioning operators take a collection, produces a set (one per group) of sub-collections (with the same type as the input one) and applies the subsequent operation to each sub-collection. Partitioning operators are defined according to the following grammar, where π is a user-defined partitioning policy that maps each item to the respective sub-collection:

$\langle \textit{p-operator} \rangle$ ::= `p-`$\langle \textit{core-operator} \rangle$ π

Operators in the `combine`, `b-map` and `b-combine` families support partitioning, so, for instance, a `p-combine` produces a bag of values, each being the synthesis of one group; also the natural join operator from the relational algebra is a particular case of per-group joining.

The decomposition by both partitioning and windowing considers the former as the external decomposition, thus it logically produces a set (one per group) of collections of windows:

$\langle \textit{w-p-operator} \rangle$ ::= `w-p-`$\langle \textit{core-operator} \rangle$ π ω

2.3 Running Example: The word-count Pipeline

Algorithm 1 A WORD-COUNT Pipeline

```
$f = \lambda l.$list-map $(\lambda w.\,(w, 1))$ (split $l$)
tokenize = flatmap $f$

$\oplus = \lambda xy.\,(\pi_1(x), \pi_2(x) + \pi_2(y))$
keyed-sum = p-(reduce $\oplus$) $\pi_1$

file-read = from-file input-file
file-write = to-file output-file

WORD-COUNT = NEW tokenize | NEW keyed-sum
FILE-WORD-COUNT = NEW file-read | WORD-COUNT | NEW file-write
```

We illustrate a simple WORD-COUNT Pipeline in Algorithm 1. We assume an hypothetical PiCo implementation where the host language provides some common functions over basic types—such as strings and lists—and a syntax for defining and naming functional transformations. In this setting, the functions f and $\oplus$ in the example are user-defined kernels (i.e., functional transformations) and:

- split is a host function mapping a text line (i.e., a string) into the list of words occurring in the line
- list-map is a classical host map over lists

- π_1 is the left-projection partitioning policy (cf. example below, Section 4.1, Definition 3)

The operators have the following meaning:

- `tokenize` is a `flatmap` operator that receives lines l of text and produces, for each word w in each line, a pair $(w, 1)$;
- `keyed-sum` is a `p-reduce` operator that partitions the pairs based on w (obtained with π_1, using group-by-word) and then sums up each group to (w, n_w), where w occurs n_w times in the input text;
- `file-read` is an `emit` operator that reads from a text file and generates a list of lines;
- `file-write` is a `collect` operator that writes a bag of pairs (w, n_w) to a text file.

3 Type System

Legal Pipelines are defined according to typing rules, described below. We denote the typing relation as $a : \tau$, if and only if there exists a legal inference assigning type τ to the term a.

3.1 Collection Types

We mentioned earlier (Section 2) that collections are *implicit* entities that flow across Pipelines through the DAG edges. A collection is either *bounded* or *unbounded*; moreover, it is also either *ordered* or *unordered*. A combination of the mentioned characteristics defines the *structure type* of a collection. We refer to each structure type with a mnemonic name:

- a bounded, ordered collection is a *list*
- a bounded, unordered collection is a (bounded) *bag*
- an unbounded, ordered collection is a *stream*

A collection type is characterized by its structure type and its *data type*, namely the type of the collection elements. Formally, a collection type has form T_σ where $\sigma \in \Sigma$ is the structure type, T is the data type—and where $\Sigma = \{\text{bag}, \text{list}, \text{stream}\}$ is the set of all structure types. We also partition Σ into Σ_b and Σ_u, defined as the sets of bounded and unbounded structure types, respectively. Moreover, we define Σ_o as the set of ordered structure types, thus $\Sigma_b \cap \Sigma_o = \{\text{list}\}$ and $\Sigma_u \cap \Sigma_o = \{\text{stream}\}$. Finally, we allow the void type $\emptyset$.

Operator	Type
Unary	
`map`	$T_\sigma \to U_\sigma, \forall\sigma \in \Sigma$
`combine`, `p-combine`	$T_\sigma \to U_\sigma, \forall\sigma \in \Sigma_b$
`w-combine`, `w-p-combine`	$T_\sigma \to U_\sigma, \forall\sigma \in \Sigma_o$
`emit`	$\emptyset \to U_\sigma$
`collect`	$T_\sigma \to \emptyset$
Binary	
`b-map`, `p-b-map`	$T_\sigma \times T'_\sigma \to U_\sigma, \forall\sigma \in \Sigma_b$
`w-b-map`, `w-p-b-map`	$T_\sigma \times T'_\sigma \to U_\sigma, \forall\sigma \in \Sigma_o$

Table 3: Operator types.

$$\frac{op : T_\sigma \to U_\sigma, \sigma \in \Sigma_o}{\texttt{w-}op\ \omega : T_{\sigma'} \to U_{\sigma'}, \sigma' \in \Sigma_o}\ \texttt{w-}$$

Figure 2: Unbounded extension provided by windowing

3.2 Operator Types

Operator types are defined in terms of input/output signatures. The typing of operators is reported in Table 3. We do not show the type inference rules since they are straightforward.

From the type specification, we say each operator is characterized by its input and output degrees (i.e., the cardinality of left and right-hand side of the $\to$ symbol, respectively). All operators but `collect` have output degree 1, while `collect` has output degree 0. All binary operators have input degree 2, `emit` has input degree 0 and all the other operators have input degree 1.

All operators are polymorphic with respect to data types. Moreover, all operators but `emit` and `collect` are polymorphic with respect to structure types. Conversely, each `emit` and `collect` operator deals with one specific structure type.[4]

As we mentioned in Section 3.1, a windowing operator may behave as the unbounded extension of the respective bounded operator. This is formalized by the inference rule `w-` that is reported in Figure 2: given an operator *op* dealing with ordered structure types (bounded or unbounded), its windowing counterpart `w-`*op* can operate on *any* ordered structure type, including stream. The analogous principle underlies the inference rules for all the `w-` operators.

[4]For example, an emitter for a finite text file would generate a bounded collection of strings, whereas an emitter for stream of tweets would generate an unbounded collection of tweet objects.

$$\frac{op : \tau}{\text{NEW}\ op : \tau}\ \text{NEW}$$

$$\frac{p : T^{\circ}_{\sigma} \to U_{\sigma} \quad p_i : U_{\sigma} \to (V^{\circ}_{\sigma})_i \quad \exists i : (V^{\circ}_{\sigma})_i = V_{\sigma}}{\text{TO}\ p\ p_1\ \ldots\ p_n : T^{\circ}_{\sigma} \to V_{\sigma}}\ \text{TO}$$

$$\frac{p : T^{\circ}_{\sigma} \to U_{\sigma} \quad p_i : U_{\sigma} \to \emptyset}{\text{TO}\ p\ p_1\ \ldots\ p_n : T^{\circ}_{\sigma} \to \emptyset}\ \text{TO}_{\emptyset}$$

$$\frac{p : T^{\circ}_{\sigma} \to U_{\sigma} \quad p' : \emptyset \to U'_{\sigma} \quad a : U_{\sigma} \times U'_{\sigma} \to V^{\circ}_{\sigma}}{\text{PAIR}\ p\ p'\ a : T^{\circ}_{\sigma} \to V^{\circ}_{\sigma}}\ \text{PAIR}$$

$$\frac{p : \emptyset \to U_{\sigma} \quad p' : T^{\circ}_{\sigma} \to U'_{\sigma} \quad a : U_{\sigma} \times U'_{\sigma} \to V^{\circ}_{\sigma}}{\text{PAIR}\ p\ p'\ a : T^{\circ}_{\sigma} \to V^{\circ}_{\sigma}}\ \text{PAIR}'$$

$$\frac{p : T^{\circ}_{\sigma} \to U_{\sigma} \quad p' : \emptyset \to U_{\sigma}}{\text{MERGE}\ p\ p' : T^{\circ}_{\sigma} \to U_{\sigma}}\ \text{MERGE}$$

Figure 3: Pipeline typing

3.3 Pipeline Types

Pipeline types are defined according to the inference rules in Figure 3. For simplicity, we use the meta-variable T°_{σ}, which can be rewritten as either T_{σ} or $\emptyset$, to represent the optional collection type[5]. The awkward rule TO covers the case in which, in a TO Pipeline, at least one destination Pipeline p_i has non-void output type V_{σ}; in such case, all the destination Pipelines with non-void output type must have the same output type V_{σ}, which is also the output type of the resulting Pipeline.

Finally, we define the notion of top-level Pipelines, representing Pipelines that may be executed.

Definition 1. *A* top-level *Pipeline is a* non-empty *Pipeline of type* $\emptyset \to \emptyset$.

Running Example: Typing of word-count

We present the types of the WORD-COUNT components, defined in Section 2. We omit full type derivations since they are straightforward applications of the typing rules.

[5]We remark the optional collection type is a mere syntactic rewriting, thus it does not represent any additional feature of the typing system.

The operators are all unary and have the following types:

$$\begin{array}{ll}
\texttt{tokenize} & : \text{String}_\sigma \to (\text{String} \times \mathbb{N})_\sigma, \forall\sigma \in \Sigma \\
\texttt{keyed-sum} & : (\text{String} \times \mathbb{N})_\sigma \to (\text{String} \times \mathbb{N})_\sigma, \forall\sigma \in \Sigma \\
\texttt{file-read} & : \emptyset_{\text{bag}} \to \text{String}_{\text{bag}} \\
\texttt{file-write} & : (\text{String} \times \mathbb{N})_{\text{bag}} \to \emptyset_{\text{bag}}
\end{array}$$

Pipelines have the following types:

$$\begin{array}{ll}
\textsc{word-count} & : \text{String}_\sigma \to (\text{String} \times \mathbb{N})_\sigma, \forall\sigma \in \Sigma \\
\textsc{file-word-count} & : \emptyset \to \emptyset
\end{array}$$

We remark that WORD-COUNT is polymorphic whereas FILE-WORD-COUNT is a top-level Pipeline.

4 Semantics

We propose an interpretation of Pipelines in terms of semantic Dataflow graphs, as defined in [8]. Namely, we propose the following mapping:

- Collections $\Rightarrow$ Dataflow tokens
- Operators $\Rightarrow$ Dataflow vertexes
- Pipelines $\Rightarrow$ Dataflow graphs

Note that collections in semantic Dataflow graphs are treated as a whole, thus they are mapped to single Dataflow tokens that flow through the graph of transformations. In this setting, semantic operators (i.e., Dataflow vertexes) map an input collection to the respective output collection upon a single firing.

4.1 Semantic Collections

Dataflow tokens are data collections of T-typed elements, where T is the data type of the collection. Unordered collections are semantically mapped to multi-sets, whereas ordered collections are mapped to sequences.

We denote an unordered data collection of data type T with the following, "{ ... }" being interpreted as a multi-set (i.e., unordered collection with possible multiple occurrences of elements):

$$m = \left\{m_0, m_1, \dots, m_{|m|-1}\right\} \tag{1}$$

A sequence (i.e., semantic ordered collection) associates a numeric *timestamp* to each item, representing its temporal coordinate, in time units, with respect to time zero. Therefore, we denote the generic item of a sequence having data type T as (t_i, s_i) where $i \in \mathbb{N}$ is the position of the item in the sequence, $t_i \in \mathbb{N}$ is the timestamp and $s_i \in T$ is the item value. We denote an ordered data collection of data type T with the following, where $\overset{(b)}{=}$ holds only for bounded sequences (i.e., lists):

$$\begin{aligned} s &= [(t_0, s_0), (t_1, s_1), (t_2, s_2), \ldots \bullet t_i \in \mathbb{N}, s_i \in T] \\ &= [(t_0, s_0)] +\!\!+ [(t_1, s_1), (t_2, s_2), \ldots] \\ &= (t_0, s_0) :: [(t_1, s_1), (t_2, s_2), \ldots] \\ &\overset{(b)}{=} \left[(t_0, s_0), (t_1, s_1), \ldots, (t_{|s|-1}, s_{|s|-1})\right] \end{aligned} \tag{2}$$

The symbol $++$ represents the concatenation of sequence $[(t_0, s_0)]$ (head sequence) with the sequence $[(t_1, s_1), (t_2, s_2), \ldots]$ (tail sequence). The symbol $::$ represents the concatenation of element (t_0, s_0) (head element) with the sequence $[(t_1, s_1), (t_2, s_2), \ldots]$ (tail sequence).

We define the notion of *time-ordered sequences*.
Definition 2. *A sequence $s = [(t_0, s_0), (t_1, s_1), (t_2, s_2), \ldots]$ is time-ordered when the following condition is satisfied for any $i, j \in \mathbb{N}$:*

$$i \leq j \Rightarrow t_i \leq t_j$$

We denote as $\overrightarrow{s}$ any time-ordered permutation of s. The ability of dealing with non-time-ordered sequences, which is provided by PiCo, is sometimes referred as *out-of-order* data processing [4].

Before proceeding to semantic operators and Pipelines, we define some preliminary notions about the effect of partitioning and windowing over semantic collections.

4.1.1 Partitioned Collections

In Section 2.2, we introduced partitioning policies. In semantic terms, a partitioning policy π defines how to group collection elements.
Definition 3. *Given a multi-set m of data type T, a function $\pi : T \rightarrow K$ and a key $k \in K$, we define the k-selection $\sigma_k^\pi(m)$ as follows:*

$$\sigma_k^\pi(m) = \{m_i \bullet x \in m_i \land \pi(m_i) = k\} \tag{3}$$

Similarly, the k-selection $\sigma_k^\pi(s)$ of a sequence s is the sub-sequence of s such that the following holds:

$$\forall (t_i, s_i) \in s, (t_i, s_i) \in \sigma_k^\pi(s) \iff \pi(s_i) = k \tag{4}$$

We define the partitioned collection as the set of all groups generated according to a partitioning policy.

Definition 4. *Given a collection c and a partitioning policy π, the partitioned collection c according to π, noted $c^{(\pi)}$, is defined as follows:*

$$c^{(\pi)} = \{\sigma_k^\pi(c) \bullet k \in K \wedge |\sigma_k^\pi(c)| > 0\} \tag{5}$$

We remark that partitioning has no effect with respect to time-ordering.

Example: The group-by-key decomposition, with π_1 being the left projection,[6] uses a special case of selection where:

- the collection has data type $K \times V$
- $\pi = \pi_1$

4.1.2 Windowed Collections

Before proceeding further, we provide the preliminary notion of *sequence splitting*. A splitting function f defines how to split a sequence into two possibly overlapping sub-sequences, namely the *head* and the *tail*.

Definition 5. *Given a sequence s and a splitting function f, the splitting of s according to f is:*

$$f(s) = (h(s), t(s)) \tag{6}$$

where $h(s)$ is a bounded prefix of s, $t(s)$ is a proper suffix of s, and there is a prefix p of $h(s)$ and a suffix u of $t(s)$ such that $s = p +\!\!+ u$.

In Section 2.2.4, we introduced windowing policies. In semantic terms, a windowing policy ω identifies a splitting function $f^{(\omega)}$. Considering a split sequence $f_\omega(s)$, the head $h_\omega(s)$ represents the elements falling into the window, whereas the tail $t_\omega(s)$ represents the remainder of the sequence.

We define the windowed sequence as the result of repeated applications of windowing with time-reordering of the heads.

Definition 6. *Given a sequence s and a windowing policy w, the windowed view of s according to w is:*

$$s^{(\omega)} = [\overrightarrow{s_0}, \overrightarrow{s_1}, \ldots, \overrightarrow{s_i}, \ldots] \tag{7}$$

where $s_i = h_\omega(\underbrace{t_\omega(t_\omega(\ldots t_\omega}_{i}(s)\ldots)))$

Example: The count-based policy $\omega = (5, 2, \texttt{count})$ extracts the first 5 items from the sequence at hand and discards the first 2 items of the sequence upon sliding, whereas the tumbling policy $\omega = (5, 5, \texttt{count})$ yields non-overlapping contiguous windows spanning 5 items.

[6] $\pi_1(x, y) = x$

4.2 Semantic Operators

We define the semantics of each operator in terms of its behavior with respect to token processing by following the structure of Table 3. We start from bounded operators and then we show how they can be extended to their unbounded counterparts by considering windowed streams.

Dataflow vertexes with one input edge and one output edge (i.e., unary operators with both input and output degrees equal to 1) take as input a token (i.e., a data collection), apply a transformation, and emit the resulting transformed token. Vertexes with no input edges (i.e., `emit`)/no output edges (i.e., `collect`) execute a routine to produce/consume an output/input token, respectively.

4.2.1 Semantic Core Operators

The bounded `map` operator has the following semantics:

$$\begin{aligned}\texttt{map}\, f\, m &= \{f(m_i) \bullet m_i \in m\} \\ \texttt{map}\, f\, s &= \big[(t_0, f(s_0)), \ldots, (t_{|s|-1}, f(s_{|s|-1}))\big]\end{aligned} \tag{8}$$

where m and s are input tokens (multi-set and list, respectively) whereas right-hand side terms are output tokens. In the ordered case, we refer to the above definition as *strict* semantic `map`, since it respects the global time-ordering of the input collection.

The bounded `flatmap` operator has the following semantics:

$$\begin{aligned}\texttt{flatmap}\, f\, m &= \bigcup \{f(m_i) \bullet m_i \in m\} \\ \texttt{flatmap}\, f\, s &= [(t_0, f(s_0)_0), (t_0, f(s_0)_1), \ldots, (t_0, f(s_0)_{n_0})] +\!\!+ \\ &\quad\; [(t_1, f(s_1)_0), \ldots, (t_1, f(s_1)_{n_1})] +\!\!+ \ldots +\!\!+ \\ &\quad\; \big[(t_{|s|-1}, f(s_{|s|-1})_0) \ldots, (t_{|s|-1}, f(s_{|s|-1})_{n_{|s|-1}})\big]\end{aligned} \tag{9}$$

where $f(s_i)_j$ is the j-th item of the list $f(s_i)$, that is, the output of the kernel function f over the input s_i. Notice that the timestamp of each output item is the same as the respective input item.

The bounded `reduce` operator has the following semantics, where $\oplus$ is both associative and commutative and, in the ordered variant, $t' = \max\limits_{(t_i, s_i) \in s} t_i$:

$$\begin{aligned}\texttt{reduce}\, \oplus\, m &= \{\textstyle\bigoplus \{m_i \in m\}\} \\ \texttt{reduce}\, \oplus\, s &= \big[(t', (\ldots(s_0 \oplus s_1) \oplus \ldots) \oplus s_{|s|-1})\big] \\ &\overset{(a)}{=} \big[(t', s_0 \oplus s_1 \oplus \ldots \oplus s_{|s|-1})\big] \\ &\overset{(c)}{=} [(t', \textstyle\bigoplus \Pi_2(s))]\end{aligned} \tag{10}$$

meaning that, in the ordered variant, the timestamp of the resulting value is the same as the input item having the maximum timestamp. Equation $\overset{(a)}{=}$ holds since $\oplus$ is associative and equation $\overset{(c)}{=}$ holds since it is commutative.

The `fold+reduce` operator has a more complex semantics, defined with respect to an *arbitrary* partitioning of the input data. Informally, given a partition P of the input collection, each subset $P_i \in P$ is mapped to a local accumulator a_i, initialized with value z; then:

1. Each subset P_i is folded into its local accumulator a_i, using $\oplus_1$;
2. The local accumulators a_i are combined using $\oplus_2$, producing a reduced value r;

The formal definition—that we omit for the sake of simplicity—is similar to the semantic of `reduce`, with the same distinction between ordered and unordered processing and similar considerations about associativity and commutativity of user functions. We assume, without loss of generality, that the user parameters z and $\oplus_1$ are always defined such that the resulting `fold+reduce` operator is partition-independent, meaning that the result is independent from the choice of the partition P.

4.2.2 Semantic Decomposition

Given a bounded `combine` operator *op* and a selection function $\pi : T \to K$, the partitioning operator `p-`*op* has the following semantics over a generic collection c:

$$\texttt{p-}op\ \pi\ c = \left\{ op\ c' \bullet c' \in c^{(\pi)} \right\}$$

For instance, the group-by-key processing is obtained by using the by-key partitioning policy (cf. example below definition 3).

Similarly, given a bounded `combine` operator *op* and a windowing policy ω, the windowing operator `w-`*op* has the following semantics:

$$\texttt{w-}op\ \omega\ s = op\ s_0^{(\omega)} +\!\!+ \ldots +\!\!+\ op\ s_{\left|s^{(\omega)}\right|-1}^{(\omega)} \tag{11}$$

where $s_i^{(\omega)}$ is the i-th list in $s^{(\omega)}$ (cf. Definition 6).

As for the combination of the two partitioning mechanisms, `w-p-`*op*, it has the following semantics:

$$\texttt{w-p-}op\ \pi\ \omega\ s = \left\{ \texttt{w-}op\ \omega\ s' \bullet s' \in s^{(\pi)} \right\}$$

Thus, as mentioned in Section 2.2, partitioning first performs the decomposition, and then processes each group on a per-window basis.

4.2.3 Unbounded Operators

We remark that none of the semantic operators defined so far can deal with unbounded collections. As mentioned in Section 2.2, we rely on windowing for extending them to the unbounded case.

Given a (bounded) windowing `combine` operator *op*, the semantics of its unbounded variant is a trivial extension of the bounded case:

$$\texttt{w-}op\ \omega\ s = op\ s_0^{(\omega)} +\!\!+ \ldots +\!\!+\ c\ s_i^{(\omega)} +\!\!+ \ldots \tag{12}$$

The above incidentally also defines the semantics of unbounded windowing and partitioning `combine` operators.

We rely on the analogous approach to define the semantics of unbounded operators in the `map` family, but in this case the windowing policy is introduced at the semantic rather than syntactic level, since `map` operators do not support decomposition. Moreover, the windowing policy is forced to be batching (cf. Example below Definition 5). We illustrate this concept on `map` operators, but the same holds for `flatmap` ones. Given a bounded `map` operator, the semantics of its unbounded extension is as follows, where ω is a tumbling windowing policy:

$$[\![\texttt{map}\ f\ s]\!]_\omega = \texttt{map}\ f\ s_0^{(\omega)} +\!\!+ \ldots +\!\!+\ \texttt{map}\ f\ s_i^{(\omega)} +\!\!+ \ldots \tag{13}$$

We refer to the above definition as *weak* semantic `map` (cf. strict semantic `map` in Equation 8), since the time-ordering of the input collection is partially dropped. In the following chapters, we provide a PiCo implementation based on weak semantic operators for both bounded and unbounded processing.

4.2.4 Semantic Sources and Sinks

Finally, `emit`/`collect` operators do not have a functional semantics, since they produce/consume collections by interacting with the system state (e.g., read-/write from/to a text file, read/write from/to a network socket). From the semantic perspective, we consider each `emit`/`collect` operator as a Dataflow node able to produce/consume as output/input a collection of a given type, as shown in Table 3. Moreover, `emit` operators of ordered type have the responsibility of tagging each emitted item with a timestamp.

4.3 Semantic Pipelines

The semantics of a Pipeline maps it to a semantic Dataflow graph. We define such mapping by induction on the Pipeline grammar defined in Section 2. The following definitions are basically a formalization of the pictorial representation in Figure 1.

We also define the notion of *input*, resp. *output*, vertex of a Dataflow graph G, denoted as $v_I(G)$ and $v_O(G)$, respectively. Conceptually, an input node represents a Pipeline source, whereas an output node represents a Pipeline sink.

The following formalization provides the semantics of any PiCo program.

- (NEW *op*) is mapped to the graph $G = (\{op\}, \emptyset)$; moreover, one of the following three cases hold:
  - *op* is an `emit` operator, then $v_O(G) = op$, while $v_I(G)$ is undefined
  - *op* is a `collect` operator, then $v_I(G) = op$, while $v_O(G)$ is undefined
  - *op* is an unary operator with both input and output degree equal to 1, then $v_I(G) = v_O(G) = op$
- (TO $p\ p_1\ \ldots\ p_n$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V = &\ V(G_p) \cup V(G_{p_1}) \cup \ldots \cup V(G_{p_n}) \cup \{\mu\} \\ E = &\ E(G_p) \cup \textstyle\bigcup_{i=1}^{n} E(G_{p_i}) \cup \bigcup_{i=1}^{n} \{(v_O(G_p), v_I(G_{p_i}))\} \cup \\ &\ \textstyle\bigcup_{i=1}^{|G'|} \{(v_O(G'_i), \mu)\} \end{aligned}$$

  where μ is a non-determinate merging node as defined in [7] and $G' = \{G_{p_i} \bullet d_O(G_{p_i}) = 1\}$; moreover, $v_I(G) = v_I(G_p)$ if $d_I(G_p) = 1$ and undefined otherwise, while $v_O(G) = \mu$ if $|G'| > 0$ and undefined otherwise.
- (PAIR $p\ p'\ op$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V &= V(G_p) \cup V(G_{p'}) \cup \{o\}\, p \\ E &= E(G_p) \cup E(G_{p'}) \cup \{(v_O(G_p), op)\,, (v_O(G_{p'}), op)\} \end{aligned}$$

  moreover, $v_O(G) = op$, while one of the following cases holds:
  - $v_I(G) = v_I(G_p)$ if the input degree of p is 1
  - $v_I(G) = v_I(G_{p'})$ if the input degree of p' is 1
  - $v_I(G)$ is undefined if both p and p' have output degree equal to 0
- (MERGE $p\ p'$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V &= V(G_p) \cup V(G_{p'}) \cup \{\mu\} \\ E &= E(G_p) \cup E(G_{p'}) \cup \{(v_O(G_p), \mu)\,, (v_O(G_{p'}), \mu)\} \end{aligned}$$

  where μ is a non-determinate merging node; moreover, $v_O(G) = \mu$, while one of the following cases holds:
  - $v_I(G) = v_I(G_p)$ if the input degree of p is 1
  - $v_I(G) = v_I(G_{p'})$ if the input degree of p' is 1
  - $v_I(G)$ is undefined if both p and p' have output degree equal to 0

Running Example: Semantics of word-count

The tokens (i.e., data collections) flowing through the semantic Dataflow graph resulting from the WORD-COUNT Pipeline are bags of strings (e.g., lines produced by `file-read` and consumed by `tokenize`) or bags of string-$\mathbb{N}$ pairs (e.g., counts produced by `tokenize` and consumed by `keyed-sum`). In this example, as usual, string-$\mathbb{N}$ pairs are treated as key-value pairs, where keys are strings (i.e., words) and values are numbers (i.e., counts).

By applying the semantic of `flatmap`, `reduce` and `p-(reduce` $\oplus$) to Algorithm 1, the result obtained is that the token being emitted by the `combine` operator is a bag of pairs (w, n_w) for each word w in the input token of the `flatmap` operator.

The Dataflow graph resulting from the semantic interpretation of the WORD-COUNT Pipeline defined in Section 2 is $G = (V, E)$, where:

$$\begin{aligned} V &= \{\texttt{tokenize}, \texttt{keyed-sum}\} \\ E &= \{(\texttt{tokenize}, \texttt{keyed-sum})\} \end{aligned}$$

Finally, the FILE-WORD-COUNT Pipeline results in the graph $G = (V, E)$ where:

$$\begin{aligned} V &= \{\texttt{file-read}, \texttt{tokenize}, \texttt{keyed-sum}, \texttt{file-write}\} \\ E &= \{(\texttt{file-read}, \texttt{tokenize}), \\ &\quad (\texttt{tokenize}, \texttt{keyed-sum}), \\ &\quad (\texttt{keyed-sum}, \texttt{file-write})\} \end{aligned}$$

5 Programming Model Expressiveness

In this section, we provide a set of use cases adapted from examples in Flink's user guide [3]. Besides they are very simple examples, they exploit grouping, partitioning, windowing and Pipelines merging. We aim to show the expressiveness of our model without using any concrete API, to demonstrate that the model is independent from its implementation.

5.1 Use Cases: Stock Market

The first use case is about analyzing stock market data streams. In this use case, we:

1. read and merge two stock market data streams from two sockets (algorithm 2)
2. compute statistics on this market data stream, like rolling aggregations per stock (algorithm 3)
3. emit price warning alerts when the prices change (algorithm 4)
4. compute correlations between the market data streams and a Twitter stream with stock mentions (algorithm 5)

Algorithm 2 The READ-PRICE Pipeline

```
READ-PRICES = NEW from-socket $s_1$ + NEW from-socket $s_2$
```

Read from multiple sources Algorithm 2 shows the STOCK-READ Pipeline, which reads and merges two stock market data streams from sockets s_1 and s_2. Assuming StockName and Price are types representing stock names and prices, respectively, then the type of each `emit` operator is the following (since `emit` operators are polymorphic with respect to data type):

$$\emptyset \rightarrow (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$$

Therefore it is also the type of READ-PRICES since it is a MERGE of two `emit` operators of such type.

Algorithm 3 The STOCK-STATS Pipeline

```
min = reduce $(\lambda xy.\min(x, y))$
max = reduce $(\lambda xy.\max(x, y))$
sum-count = fold+reduce  $(\lambda ax.((\pi_1(a)) + 1, (\pi_2(a)) + x))\ (0, 0)$
                         $(\lambda a_1 a_2.(\pi_1(s_1) + \pi_1(a_2), \pi_2(a_1) + \pi_2(a_2)))$
normalize = map $(\lambda x.\pi_2(x)/\pi_1(x))$
$\omega = (10, 5,$ count$)$

STOCK-STATS = TO  READ-PRICES
                  NEW w-p-(min) $\pi_1\ \omega$
                  NEW w-p-(max) $\pi_1\ \omega$
                  (NEW w-p-(sum-count) $\pi_1\ \omega$ | NEW normalize)
```

Statistics on market data stream Algorithm 3 shows the STOCK-STATS Pipeline, that computes three different statistics—minimum, maximum and mean—for each stock name, over the prices coming from the READ-PRICES Pipeline. These statistics are windowing based, since the data processed belongs to a stream possibly unbound. The specified window policy $\omega = (10, 5, \texttt{count})$ creates windows of 10 elements with sliding factor 5.

The type of STOCK-STATS is $\emptyset \rightarrow (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$, the same as READ-PRICES.

Algorithm 4 The PRICE-WARNINGS Pipeline

```
collect = fold+reduce  $(\lambda s x. s \cup \{x\})\ \emptyset$
                       $(\lambda s_1 s_2. s_1 \cup s_2)$
fluctuation = map $(\lambda s.\text{set-fluctuation}(s))$
high-pass = flatmap $(\lambda \delta.\text{if } \delta \geq 0.05 \text{ then yield } \delta)$
$\omega = (10, 5, \texttt{count})$

PRICE-WARNINGS = READ-PRICES |
                 NEW w-p-(collect) $\pi_1$ $\omega$ | NEW fluctuation
                 NEW high-pass
```

Generate price fluctuation warnings Algorithm 4 shows the Pipeline PRICE-WARNINGS, that generates a warning each time the stock market data within a window exhibits high price fluctuation for a certain stock name—`yield` is a host-language method that produces an element.

In the example, the `fold+reduce` operator `fluctuation` just builds the sets, one per window, of all items falling within the window, whereas the downstream `map` computes the fluctuation over each set. This is a generic pattern that allows to combine collection items by re-using available user functions defined over collective data structures.

The type of PRICE-WARNINGS is again $\emptyset \rightarrow (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$.

Algorithm 5 The CORRELATE-STOCKS-TWEETS Pipeline

```
READ-TWEETS = NEW from-twitter | NEW tokenize-tweets
$\omega = (10, 10, \texttt{count})$

CORRELATE-STOCKS-TWEETS = PAIR PRICE-WARNINGS READ-TWEETS
                               w-p-(correlate) $\pi_1$ $\omega$
```

Correlate warnings with tweets Algorithm 5 shows CORRELATE-STOCKS-TWEETS, a Pipeline that generates a correlation between warning generated by PRICE-WARNINGS and tweets coming from a Twitter feed. The READ-TWEETS Pipeline generates a stream of (StockName $\times$ String) items, representing tweets each mentioning a stock name. Stocks and tweets are paired according to a join-by-key policy (cf. definition 3), where the key is the stock name.

In the example, `correlate` is a `join-fold+reduce` operator that computes the correlation between two joined collections. As we mentioned in Section 2.2, we rely on windowing to apply the (bounded) `join-fold+reduce` operator to unbounded streams. In the example, we use a simple tumbling policy $\omega = (10, 10, \texttt{count})$ in order to correlate items from the two collections in a 10-by-10 fashion.

6 Conclusion

We proposed a new programming model based on Pipelines and operators, which are the building blocks of PiCo programs, first defining the syntax of programs, then providing a formalization of the type system and semantics.

The contribution of PiCo with respect to the state-of-the-art in tools for Big Data Analytics is also in the definition and formalization of a programming model that is independent from the effective API and runtime implementation. In the state-of-the-art tools for Analytics, this aspect is typically not considered and the user is left in some cases to its own interpretation of the documentation. This happens particularly when the implementation of operators in state-of-the-art tools is conditioned in part or totally by the runtime implementation itself.

Acknowledgements

This work was partly supported by the EU-funded project TOREADOR (contract no. H2020-688797), the EU-funded project Rephrase (contract no. H2020-644235), and the 2015–2016 IBM Ph.D. Scholarship program. We gratefully acknowledge Prof. Luca Padovani for his comments on the early version of the manuscript.

References

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernàndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[2] Flink. Apache Flink website. https://flink.apache.org/.

[3] Flink. Flink streaming examples, 2015. [Online; accessed 16-November-2016].

[4] Google. *Google Cloud Dataflow*, 2015. https://cloud.google.com/dataflow/.

[5] Kappa-Architecture. Kappa-Architecture website. http://milinda.pathirage.org/kappa-architecture.com/.

[6] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792, Oct 2015.

[7] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.

[8] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. In *Proc. of HLPP2016: Intl. Workshop on High-Level Parallel Programming*, pages 1–19, Muenster, Germany, July 2016. arXiv.org.

[9] C. Misale, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017.

[10] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. *CoRR*, abs/1504.00788, 2015.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX.