IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

PiCo: a Novel Approach to Stream Data Analytics

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1659344 since 2018-05-12T11:43:31Z

Publisher:

Springer

Published version:

DOI:10.1007/978-3-319-75178-8_10

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

12 October 2024

Authors copy (postprint) ot C. Misale, M. Drocco, G. Tremblay, and M. Aldinucci, "PiCo: a Novel Approach to Stream Data Analytics," in Proc. of Euro-Par Workshops: 1st Intl. Workshop on Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP 2017), Santiago de Compostela, Spain, 2018. doi:10.1007/978-3-319-75178-8_10

PiCo: a Novel Approach to Stream Data Analytics

Claudia Misale[1], Maurizio Drocco[1], Guy Tremblay[2], Marco Aldinucci[1]

[1] Department, University of Torino. Torino, Italy
{misale, drocco, aldinuc}@di.unito.it
[2] Dépt. d'Informatique, Université du Québec à Montréal. Montréal, QC, Canada
tremblay.guy@uqam.ca

Abstract. In this paper, we present a new C++ API with a fluent interface called PiCo (Pipeline Composition). PiCo's programming model aims at making easier the programming of data analytics applications while preserving or enhancing their performance. This is attained through three key design choices: 1) unifying batch and stream data access models, 2) decoupling processing from data layout, and 3) exploiting a stream-oriented, scalable, efficient C++11 runtime system. PiCo proposes a programming model based on pipelines and operators that are polymorphic with respect to data types in the sense that it is possible to re-use the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc.). Preliminary results show that PiCo can attain better performances in terms of execution times and hugely improve memory utilization when compared to Spark and Flink in both batch and stream processing.

1 Introduction

In the context of Big Data analytics, there is a series of tools aiming at simplifying programming applications to be executed on clusters. Although each tool claims to provide better programming, data and execution models—for which only informal (and often confusing) semantics are generally provided[1]—they all share some characteristics at different levels. From a high-level perspective, Big Data is about extracting knowledge from both structured and unstructured data. Extracting knowledge from Big Data requires tools satisfying strong requirements with respect to programmability and performance. The common aim of Big Data tools is to ensure ease of programming by providing a unique framework addressing both batch and stream processing. Even when 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

[1] For instance, consider Spark's `dstream.foreachRDD`, which provides access to RDDs in a DStream, declared as immutable collections of objects, accessible only with collective operators.

often having different semantics. We advocate a new API with a fluent interface (with method chaining) [11], called PiCo (**Pi**peline **Co**mposition), designed over the presented layered Dataflow conceptual framework [13,14]. PiCo programming model aims at *easing the programming* and *enhancing the performance* of Analytics applications through three design choices: 1) unifying batch and stream data access models, 2) decoupling processing from data layout, and 3) exploiting a stream-oriented, scalable, effiicient C++11 run-time system.

These design choices move further the level of abstraction in the programming and execution model achieved in mainstream approaches for Big Data analytics. For instance, Spark [18], Storm [15], Flink [10], and Google Dataflow [1] typically force the specialization of the algorithm to match the data access and layout. Specifically, data transformation functions (called *operators* in PiCo) exhibit different functional types when accessing data in different ways.

For this reason, the source code must often be revised when switching from one data model to the next. 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 different code needs to be written at the user-level. The Kappa architecture advocates the opposite approach, i.e., to "streamize" batch processing, but the streamizing proxy has to be coded. As for the Lambda architecture, it requires the implementation of both a batch-oriented and a stream-oriented algorithm, which means coding and maintaining two codebases.

PiCo fully decouples algorithm design from data model and layout. Code is designed in a fully functional style by composing stateless *operators*. As we discuss in the present paper, 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. Note that in other mainstream frameworks, such as Spark, the update of a pipeline by changing a transformation with another may not be trivial, since this may require the development of input and output proxies to adapt the new transformation for the calling context. Moreover, PiCo relies on FastFlow [3,4,9], a parallel programming framework designed to support streaming applications on cache-coherent multicore platforms.

2 Related Work

In this section, we provide background related to Big Data analytics tools from a stream processing perspective. Apache Spark design is intended to address iterative computations by reusing the working dataset by keeping it in memory [20,18,19]. For this reason, Spark represents a landmark in Big Data tools history, having a strong success in the community. The overall framework and

parallel computing model of Spark is similar to MapReduce, while the innovation is in the data model, represented by the *Resilient Distributed Dataset* (RDD). An RDD is a read-only collection of objects partitioned across a cluster of computers that can be operated on in parallel. A Spark program can be characterized by the two kinds of operations applicable to RDDs: *transformations* and *actions*. Those transformations and actions compose the directed acyclic graph (DAG) representing the application. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [20]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batches*. Operations over DStreams are "forwarded" to each RDD in the DStream, thus the semantics of operations over streams is defined in terms of batch processing according to the simple translation $\mathtt{op}(a) = [\mathtt{op}(a_1), \mathtt{op}(a_2), \ldots]$, where $[\cdot]$ refers to a possibly unbounded ordered sequence, $a = [a_1, a_2, \ldots]$ is a DStream, and each item a_i is a micro-batch of type RDD. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees.

Formerly known as Stratosphere [5], *Apache Flink* [7] focuses on stream programming. The abstraction used is the *DataStream*, which is a representation of a stream as a single object. Operations are composed (i.e., pipelined) by calling operators on DataStream objects. Flink also provides the *DataSet* type for batch applications, that identifies a single immutable multiset—a stream of one element. A Flink program, either for stream or batch processing, is a term from an algebra of operators over DataStreams or DataSets, respectively. Flink, differently from Spark, is a stream processing framework, meaning that both batch and stream processing are based on a streaming runtime. It can be considered one of the more advanced stream processors as many of its core features were already considered in the initial design [7].

Apache Storm is a framework targeting only stream processing [15,16,17]. It is perhaps the first widely used large-scale stream processing framework in the open source world. Storm's programming model is based on three key notions: *Spouts*, *Bolts*, and *Topologies*. A Spout is a source of a stream, that is (typically) connected to a data source or that can generate its own stream. A Bolt is a processing element, so it processes any number of input streams and produces any number of new output streams. A topology is a composition of Spout and Bolts.

Google Dataflow SDK [1] is part of the Google Cloud Platform [12]. Here, the term "Dataflow" is used by reference to the "Dataflow model", to describe the processing and programming model of the Cloud Platform. This framework aims at providing a unified model for stream, batch, and micro-batch processing. The base entity is the *Pipeline*, representing a data processing job consisting of a set of operations that can read a source of input data, transform that data, and write out the resulting output. The data model in Google Dataflow is represented by *PCollection*s, representing a potentially large, immutable bag of elements, that

can be either bounded or unbounded. The bounded (or unbounded) nature of a PCollection affects how Dataflow processes the data. Bounded PCollections can be processed using batch jobs, that might read the entire data set once and perform processing in a finite job. Unbounded PCollections must be processed using streaming jobs, as the entire collection may never be available for processing at any one time and they can be grouped by using windowing to create logical windows of finite size.

Thrill [6] is a prototype of a general purpose big data batch processing framework with a dataflow style programming interface implemented in C++ and exploiting template meta-programming. Thrill's data model is the *Distributed Immutable Array* (DIA), an array of items distributed over the cluster, to which no direct access to elements is permitted—i.e., it is only possible to apply operations to the array as a whole. A DIA remains an abstract entity flowing between two concrete DIA operations, allowing to apply optimizations such as pipelining or chaining, combining the logic of one or more functions into a single one (called pipeline). A consequence of using C++ is that memory has to be managed explicitly, although memory management in modern C++11 has been considerably simplified—for instance, Thrill uses reference counting extensively. Thrill provides a SPMD (Single Program, Multiple Data) execution model, similar to MPI, where the same program is run on different machines,

3 PiCo Programming Model

In this section, we present the PiCo C++ API, consisting of two main categories of elements: Pipelines and Operators—PiCo's formal semantics is described in [8]. Note that the design of the Operators API is based on inheritance, following faithfully PiCo's grammar specification [8]—even though the use of template programming without inheritance might have slightly improved the runtime performance. Thus, the implementation makes use of dynamic polymorphism when building the semantics DAG, where virtual member functions are invoked to determine the kind of Operator currently processed.

3.1 Pipe and Operators

A C++ PiCo program is a set of operator objects composed into a `Pipe` object, processing bounded or unbounded data.

A Pipeline can be: 1. created as the empty `Pipe` (default constructor); 2. created as a `Pipe` consisting of a single operator; 3. modified by adding an operator, through the `add` function; 4. modified by appending other `Pipe`s, through the `to` functions; 5. merged with another `Pipe`, through the `merge` function; 6. paired with another `Pipe` by means of a binary operator, through the `pair` function.

```
typedef KeyValue<std::string, int> KV;

static auto tokenizer = [](std::string& in,FlatMapCollector<KV>& collector) {
   std::istringstream f(in);
   std::string s;
   while (std::getline(f, s, ' ')) {
      collector.add(KV(s,1));
   }
};

int main(int argc, char** argv) {
   // Parse command line
   parse_PiCo_args(argc, argv);

   // Define a generic word-count pipeline
   Pipe countWords;
   countWords
     .add(FlatMap<std::string, std::string>(tokenizer))
     .add(Map<std::string, KV>([&](std::string in) { return KV(in,1); } ))
     .add(PReduce<KV>([&](KV v1, KV v2) { return v1+v2; } ));

   // Define I/O operators from/to file
   ReadFromFile reader();
   WriteToDisk<KV> writer([&](KV in) {
      return in.to_string();
   });

   // Compose the pipeline
   Pipe p2;
   p2
     .add(reader)
     .to(countWords) // append to...
     .add(writer);

   // Execute the pipeline
   p2.run();

   return 0;
}
```

Listing 1.1: Word Count example in PiCo.

Operators can be unary or binary. `UnaryOperator` is the base class representing PiCo unary operators, those with no more than one input and/or output collection. For instance, a `Map` object takes a C++ callable value (i.e., a kernel) as parameter and represents a PiCo operator `map`, which processes a collection by applying the kernel to each item. Also, `ReadFromFile` is a sub-class of `UnaryOperator` and represents PiCo operators that produce a (bounded) unordered collection of lines, read from an input text file.

`BinaryOperator` is the base class representing operators with two input collections and one output collection. For instance, a `BinaryMap` object represents a PiCo operator `b-map` that processes pairs of elements coming from two different input collections and produces a single output for each pair. A `BinaryMap` object is passed as parameter to Pipeline objects built by calling the `pair` member function.

Listing 1.1 shows a complete example for our Word Count benchmark.

4 Anatomy of a PiCo Application

When the `run()` member function is called on a pipeline `p1`, the semantics dataflow is processed to create the parallel execution dataflow. This latter graph represents the application in terms of processing elements (i.e., actors) connected by data channels (i.e., edges), where operators can be replicated to express data parallelism. We implemented this intermediate representation directly in FastFlow by using nodes, farms and pipelines patterns.

The creation of the parallel execution dataflow is straightforward. Having an empty `ff_pipeline picoDAG` that will be executed, we then start visiting the first node of the semantics dataflow, which can be an input or an entry point node. On the basis of its role, a new `ff_node` or `ff_farm` is instantiated and added to `picoDAG`.

The semantics DAG is recursively visited and the following operations are performed: 1. A single `ff_node` is added in case of input/output operators; 2. The corresponding `ff_farm` is added in case of operators different from I/O operators; 3. If an entry point is encountered, a new `ff_farm` is created and added to `picoDAG`: (a) a new `ff_pipeline` is created for each entry point's adjacent node; (b) these `ff_pipeline`s are built with new `ff_node`s created by recursively visiting the input Pipe's graph, until reaching the last node of each Pipe visited.

At the end, the resulting `picoDAG` is thus a composition of `ff_pipeline`s and `ff_farm`s.

4.1 FastFlow Network Execution

In this section, we describe the execution of the `picoDAG` pipeline, starting from a brief summary of the FastFlow runtime.

From the orchestration viewpoint, the process model to be employed is based on the Communicating Sequential Processes (CSP)[2] model, where processes (i.e., `ff_node`s) are named and the data paths between processes are explicitly identified (which is thus different from the Actor model). The abstract units of communication and synchronization are known as *channels* and represent a stream of data exchanged between two processes. Each `ff_node` is a C++ class entering an infinite loop through its `svc()` (*service*) member function where: 1. it gets a task from input channel; 2. it executes business code on the task; 3. it puts a task into the output channel. Representing communication and synchronization by a channel ensures that synchronization is tied to communication and allows layers of abstraction at higher levels to compose parallel programs where synchronization is implicit. Patterns to build a graph of `ff_node`s, such as farms, are defined in the *core patterns* level of the FastFlow stack. Since the graph of

[2] The CSP model describes a systems in terms of component processes operating independently, which interact with each other through message-passing communication.

`ff_node`s is a streaming network, any FastFlow graph is built using two streaming patterns (farm and pipeline) and one pattern-modifier (loopback, to build cyclic networks). As an example, we highlight the key steps during the execution of the FastFlow network of processes for a simple PiCo application with three operators: Read from File, Map, and Write to Disk. The first node is the Read from File (Rff), which reads lines from a file that are then forwarded to their following node of the pipeline. Tokens are sent out at microbatch granularity (in this case, a microbatch is a fixed size array of lines read from the input file). Since also a fixed size dataset is streamized, the Rff node reads the text file and sends out microbatches until the EOF is reached. The next stage of `Rff`s is the Emitter of the `map` farm, which processes stream of microbatches. Each worker of the `map` `ff_farm` processes the received microbatch by applying the user-defined function. Then each worker allocates a new microbatch to store the result of the user-defined function, and then deletes the received microbatch. The new microbatch is forwarded to the next node. The general behavior of a worker during its `svc()` call is that it deletes each input microbatch (allocated by the Emitter) after it has been processed and the results of the kernel function (applied to all elements of the microbatch) are stored into a new microbatch. When the Collector receives *PICO_EOS* tokens from all workers—a token specifying that the stream is finished and that there are no more tokens to process (i.e., end of file or socket closed)—it then forwards the token to the next stage, namely the Write to Disk (Wtd) node. This last node is a single sequential `ff_node`—input and output processing nodes are always sequential—writing the received data to a specified file. When the Wtd node receives *PICO_EOS*, the file is closed and the computation terminates.

5 Experiments

We compare PiCo to Flink v1.2.0 and Spark v2.1.0, focusing on expressiveness of the programming model and on performances in shared memory. We tested PiCo with both batch and stream applications. A first set of experiments were made of the following two applications: word count and stock market analysis.

Word count is considered as the "Hello, World!" of Big Data analytics, typically an example of batch processing. The input is a text file, which is first split into lines. Then, each line is tokenized into a sequence of words: this is implemented using `flatmap`, as each line may contain varying numbers of words. Each of these words from the input file are processed by a `map` operator that produces a key-value pair $\langle w, 1\rangle$ for each word w. After all words have been processed, the pairs are grouped by the word from each pair, and then the values (i.e., the 1s) are reduced by a sum. The final result is a single pair for each word, where the value represents the number of occurrences of the word in the text. (See also Listing 1.1.) As for the stock market analysis, it implements the "Stock Pricing" program computing a price for each option read from a text file. Each line is parsed to extract stock names followed by stock option data. A `map` operator

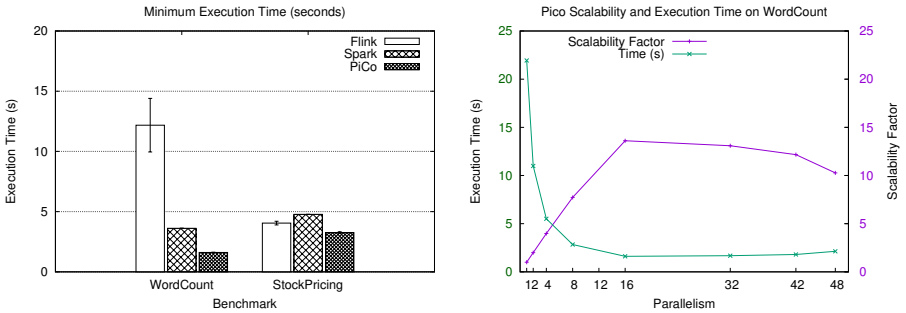

Fig. 1: (Left) Comparison of best execution times for Word Count and Stock Pricing obtained by Spark, Flink, and Pico. (Right) Scalability and execution time for Word Count in PiCo.

then computes prices by means of the Black & Scholes algorithm for each option and, finally, a reducer extracts the maximum price for each stock name.

The architecture used for experiments is the Occam Supercomputer (Open Computing Cluster for Advanced data Manipulation) [2], designed and managed by the University of Torino and the National Institute for Nuclear Physics. We used one node having the following characteristics. At hardware side: 4x Intel® Xeon® Processor E7-4830 v3 12 core/2.1Ghz, 768GB/1666MHz (48 x 16GB) DDR4 RAM, 1x SSD 800GB + 1x HDD 2TB/7200rpm, InfiniBand 56Gb + 2x Ethernet 10Gb. At software side: Linux CentOS v7.3 with Linux kernel 3.10, gcc v4.8.5 compiler (PiCo has been compiled with `O3` optimization flag), and OpenJDK Server v1.8 Java runtime.

5.1 Batch Applications

The size of the input file for the Word Count application is 600MB. It is a text file containing random words taken from a dictionary of 1K words. In the Word Count pipeline, PiCo instantiates a total of 5 fixed threads (corresponding to sequential operators), plus the main thread, plus a user-defined number of workers for the `flatmap` operator. To exploit at most 48 physical cores, we can run at most 42 worker threads. We provide a comparison on minimum execution time obtained by each tool as the average of 20 runs for each application. For the Stock Pricing application, the size of the input file is 10MB.

Figure 1 (left) shows that PiCo obtains the best execution times when compared to Spark and Flink, for both the Word Count and Stock Pricing applications. Figure 1 (right) shows scalability and execution times for the Word Count application: each value represents the average of 20 runs for each number of workers, the microbatch size is 512, and the thread pinning strategy is physical cores first.

5.2 Stream Applications

In this set of experiments, we compare PiCo to Flink and Spark when executing a stream application, the Stock Pricing one. The application is similar to the one from the batch experiment, except we added two additional option pricing algorithms—Binomial Tree and Explicit Finite Difference—and the data comes from a socket, not from a text file.

In the Stock Pricing pipeline, PiCo first instantiates 6 threads corresponding to sequential operators, such as read from socket and write to standard output, plus Emitter and Collector threads for `map` and `p-reduce` operators. Then, there is the main thread, and then k (a user-specified number) workers for the `map` and k for the `w-p-reduce` operators. With 16 workers for the `map` and 16 workers `w-p-reduce` operator mapped on physical cores, PiCo obtains the best average execution time of 7.348 seconds and a scalability factor of 14.87. We compared PiCo to Flink and Spark on the Stock Pricing streaming application. The window is count-based (or tumbling) and has size 8 in Flink and PiCo. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream*. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batches*. Operations over DStreams are "forwarded" to each RDD in the DStream, thus the semantics of operations over streams is defined in terms of batch processing. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees. Hence, Spark implements its stream processing runtime over the batch processing one, thus exploiting the BSP runtime on stream microbatches, without providing a concrete form of pipelining and reducing the real-time processing feature.

Table 1 presents the best execution times obtained by each tool, showing that PiCo obtains the best execution time and with a higher scalability compared to other tools, with a scalability of 14.87 in PiCo while 9.21 for Flink and 2.24 for Spark. Let us stress that the comparison with Spark is not completely fair since windowing is not performed in a count-based fashion. Table 1 also shows that PiCo processes more than 1.3M stock options per second, outperforming Flink and Spark, as they processes approx. 400K and 200K stock options per second respectively.

6 Conclusions

In this paper, we presented PiCo, a new C++ API with a fluent inteface for data analytics pipelines.

One key feature of PiCo is that the data model is hidden to the programmer, thus making it possible to create a model that is *polymorphic* with respect to the

Table 1: Flink, Spark and PiCo performance on Stream Stock Pricing. The execution time is the best average on 20 runs, For the same configuration, also the scalability (against Parallelism 1) and the sustained throughput are reported.

Throughput Values for 10M Stock Options				
	Execution time (s)	**Parallelism**	**Throughput (stocks/s)**	**Scalability**
Flink	24.78	16	403476.35	9.21
Spark	42.22	16	236875.81	2.24
PiCo	7.35	16	1360806.94	14.87

data model as well as to the processing model (i.e., stream or batch processing). This make it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., stream, lists, sets, etc.); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context. These aspects are fundamental to PiCo, differentiating it from other frameworks exposing different data types to be used in the same application, forcing the user to re-think the whole application when moving from one operation to another.

We compared PiCo to Flink and Spark, focusing on expressiveness of the programming model and on performances in shared memory. The current (preliminary) experiments were performed on shared memory only. By comparing execution times in both batch and stream applications, PiCo attained the best execution time when compared to two state-of-the-art frameworks, Spark and Flink. However, an aspect not mentioned above is that those experiments showed high dynamic allocation contention in input generation nodes, thus limiting PiCo scalability, a problem that will be addressed in future work. Also, results for stream processing showed that PiCo processes more than 1.3M stock options per second, outperforming Flink and Spark, that process about 400K and 200K stock options per second respectively.

Acknowledgements

This work has been partially supported by the OptiBike experiment of the EU-H2020-IA "Fortissimo2" project (no. 680481), the EU-H2020-RIA "Rephrase" project (no. 644235), the EU-H2020-RIA "Toreador" project (no. 688797), and the 2015-2016 IBM Ph.D. Scholarship program.

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. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.
2. M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino. The Open Computing Cluster for Advanced data Manipulation (OCCAM). In *Journal of Physics: Conf. Series 898 (CHEP 2016)*, San Francisco, USA, 2017.
3. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2017.
4. M. Aldinucci, M. Danelutto, M. Meneghin, M. Torquati, and P. Kilpatrick. *Efficient streaming applications on multi-core with FastFlow: The biosequence alignment test-bed*, volume 19 of *Advances in Parallel Computing*. Elsevier, 2010.
5. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.
6. T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. *CoRR*, abs/1608.05634, 2016.
7. P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.
8. M. Drocco, C. Misale, G. Tremblay, and M. Aldinucci. A formal semantics for data analytics pipelines. https://arxiv.org/abs/1705.01629, May 2017.
9. Fastflow website. `http://mc-fastflow.sourceforge.net/`, 2017 (last accessed).
10. Flink. Apache Flink website. `https://flink.apache.org/`, 2017 (last accessed).
11. M. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2011.
12. Google cloud dataflow. `https://cloud.google.com/dataflow/`, 2017 (last accessed).
13. C. Misale. *PiCo: A Domain-Specific Language for Data Analytics Pipelines*. PhD thesis, Computer Science Department, University of Torino, May 2017.
14. 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.
15. 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.
16. Storm. Apache Storm website. `http://storm.apache.org/`, 2017 (last accessed).
17. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.
18. 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.
19. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on*

Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

20. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 423–438, New York, NY, USA, 2013. ACM.