### Languages for Big Data analysis

**This is the author's manuscript**

*Original Citation:*

**Availability:**

This version is available [http://hdl.handle.net/2318/1668051](http://hdl.handle.net/2318/1668051) since 2019-03-23T09:43:44Z

**Publisher:**

Springer

**Published version:**

DOI:10.1007/978-3-319-63962-8_142-1

**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)
Languages and Frameworks for Big Data Analysis

Marco Aldinucci, Maurizio Drocco, Claudia Misale, and Guy Tremblay

Overview

Boosted by Big Data popularity, new languages and frameworks for data analytics are appearing at an increasing pace. Each of them introduces its own concepts and terminology and advocates a (real or alleged) superiority in terms of performances or expressiveness against predecessors. In this hype, for a user approaching Big Data analytics (even an educated computer scientist), it might be difficult to have a clear picture of the programming model underneath these tools and the expressiveness they provide to solve some user defined problem.

To provide some order in the world of Big Data processing, a toolkit of models to identify their common features is introduced, starting from data layout.

Data-processing applications are divided into batch vs. stream processing. Batch programs process one or more finite datasets to produce a resulting finite output dataset, whereas stream programs process possibly unbounded sequences of data, called streams, doing so in an incremental manner. Operations over streams may also have to respect a total data ordering—for instance, to represent time ordering.
In order to compare the expressiveness of programming models for Big Data analytics are mapped onto an unifying (and lower-level) computation model, i.e. the Dataflow model (Lee and Parks 1995). As shown in Misale et al (2017a), it is able to capture the distinctive features of all frameworks at all levels of abstraction, from the user-level API to the execution model. In the Dataflow model, applications as a directed graph of actors. In its “modern” macro-data flow version (Aldinucci et al 2012), it naturally models independent (thus parallelizable) kernels starting from a graph of true data dependencies, where a kernel’s execution is triggered by data availability.

The Dataflow model is expressive enough to describe batch, micro-batch and streaming models that are implemented in most tools for Big Data processing. Also, the Dataflow model helps in maturing the awareness that many Big Data analytics tools share almost the same base concepts, differing mostly in their implementation choices.

In the next section the key finding of this chapter is introduced, i.e. the Layered Dataflow Model. On this ground, in Sect. the mainstream languages and frameworks for Big Data analytics are introduced; they are Spark, Storm, Flink and Beam. According to the layered Dataflow model, they will be discussed at two successive levels of abstractions, i.e. 1) API and Semantics, and 2) Parallel Execution and Runtime Support of them will be discussed. Section draws future direction for research.

**Key Research Finding: The Layered Dataflow Model**

In order to compare different Big Data frameworks, a revised version of the layered Dataflow model is adopted (Misale et al 2017a). This model, sketched in Fig. 1, presents four layers. The top layer capture the framework API. The two intermediate layers are Dataflow models with different semantics, as described in the paragraphs below. The bottom layer is the Process Network & Platform, which embodies the network of processes used to implement a given framework together with their programming language run-time support (e.g., Java and Scala in Spark), a level which is beyond the scope of this work.

The stacked Dataflow layers are as follows.

**Framework API**

At the top of the stack, the API is the concrete primitives provided by each framework, in which data processing applications are written. Within the Big Data analyt-
Big Data Frameworks

Fig. 1 Layered model representing the levels of abstractions provided by the frameworks that were analyzed.

ics domain, APIs can be regarded as Domain-Specific Languages (DSLs), expressed in some host language (e.g., Python, Scala, Java).

**Program Semantics Dataflow**

The API exposed by all major Big Data frameworks can be explained in terms of a Dataflow graph, that is, a pair $G = (V, E)$ where actors $V$ represent operators, channels $E$ represent data dependencies among operators and tokens represent data to be processed. For instance, consider to process a collection $A$ by a function $f$ followed by a function $g$. This is represented by the graph in Fig. 2, which represents the semantic dataflow of a program computing the functional composition $f \circ g$.

![Fig. 2 Semantic dataflow graph for $f \circ g$, expressing data dependencies.](image)

Moreover, although not extensively discussed here, it is remarkable that also non-functional aspects (e.g., window-based stream processing, stateful operators, iterations) can be conveniently represented by means of the proposed Dataflow setting (Misale et al 2017a).

**Parallel Execution Dataflow**

This level represents an instantiation of the semantic dataflows in terms of processing elements (i.e., actors) connected by data channels (i.e., edges). The most
straightforward source of parallelism comes directly from the Dataflow model, namely, independent actors can run in parallel. Furthermore, some actors can be replicated to increase parallelism by making replicas work over a partition of the input data—that is, by exploiting full data parallelism. Both the above schemas are referred as embarrassingly parallel processing, since there are no dependencies among actors. Finally, in case of dependent actors that are activated multiple times (e.g. by further splitting a partition into multiple tokens), parallelism can still be exploited by letting tokens “flow” as soon as each activation is completed. This well-known schema is referred as stream/pipeline parallelism.

Process Network Dataflow & Platform

This level describes how the program is effectively deployed and executed onto the underlying platform. Actors are concrete computing entities (e.g., processes) and edges are communication channels. The most common approach is for the actual network to be a Master-Workers task executor.

Examples of Application

In this section, some mainstream frameworks for Big Data processing are discussed, within the layered Dataflow setting. Google MapReduce (Sect.) is commonly considered as the first widespread tool in this domain. Apache Spark (Sect.) provides a richer API (e.g., also targeting streams) and an implementation optimized for iterative processing. Apache Flink (Sect.) is similar to Spark, but it exploits more parallelism by means of a stream-based runtime. Apache Storm (Sect.) is focused on stream processing and, differently from the previous frameworks, Storm programs are defined as interconnected processing units, rather than functional compositions. Finally, Apache Beam (Sect.) provides an alternative API for unifying batch and stream processing under a single programming model.

Google MapReduce

Google can be considered the pioneer of Big Data processing, as the publication of the MapReduce framework paper (Dean and Ghemawat 2004) made this model mainstream. Based on the map and reduce functions, commonly used in parallel and functional programming (Cole 1989) MapReduce provides a native key-value
model and built-in sorting, that made it successful for several Big Data analytics scenarios.

API

A MapReduce program is built on the following user-defined functions: 1. a map function, that is independently applied to each item from an input key-value dataset to produce an intermediate key-value dataset; 2. a reduce function, that combines all the intermediate values associated with each key (together with the key itself) into lists of reduced values (one per key); 3. a partitioner function, that is used while sorting the intermediate dataset (i.e., before being reduced), so that the order over the key space is respected within each partition identified by the partitioner.

Figure 3 shows a source code extract from a MapReduce implementation of the Word Count application, that counts the occurrences of each word in an input text and is generally considered as the “Hello World!” for Big Data analytics. In the code extract, only map and reduce functions are specified, thus a default implementation-dependent sorting is used.

```java
1 public class WordCount {
2    3    public static class TokenizerMapper {
4        private final static IntWritable one = new IntWritable(1);
5        private Text word = new Text();
6
7        public void map(Object key, Text value, Context context)
8            throws IOException, InterruptedException {
9            StringTokenizer itr = new StringTokenizer(value.toString());
10           while (itr.hasMoreTokens()) {
11             word.set(itr.nextToken());
12             context.write(word, one);
13           }
14        }
15    }
16
17    public static class IntSumReducer {
18        private IntWritable result = new IntWritable();
19
20        public void reduce(Text key, Iterable<IntWritable> values, Context context)
21            throws IOException, InterruptedException {
22            int sum = 0;
23            for (IntWritable val : values) {
24                sum += val.get();
25            }
26            result.set(sum);
27            context.write(key, result);
28        }
29    }
30 }
```

Fig. 3 The map and reduce functions for a Java Word Count class example in MapReduce.
Semantics

The semantics of MapReduce is defined by the following chain of higher-order operators, where \( f \), \( h \), and \( \oplus \) correspond to the map, partitioner, and reduce API functions, respectively:

\[
\text{map-reduce } f \ R \oplus = (\text{flat-map } f) \circ (\text{sort } h R) \circ (\text{reduce } \oplus)
\]

When applied to an input multi-set (i.e., a finite unordered collection, possibly containing duplicated items) of key-value pairs, flat-map applies the kernel to each item and collapses all the results into a single intermediate multi-set. Formally, where \( f : K \times V \rightarrow P(K' \times V') \) is the kernel and \( m \) is a collection with \((K \times V)\)-typed items:

\[
\text{flat-map } f \ m = \bigcup \{ f(k, v) : (k, v) \in m \}
\]

The intermediate multi-set is processed by the sort operator to produce a multi-set of partially-ordered multi-sets (called intermediate partitions in MapReduce terminology). This partial sorting is parametric with respect to the number of partitions and the partitioning function, the latter mapping key-value input pairs to a partition. Formally, where \( I \) is the partition identifiers space, \( R \) is the number of partitions, and \( h : K' \times \mathbb{N} \rightarrow P \) is the partitioning function (e.g., hash-based), the partition identified by \( i \in I \) from the intermediate multi-set \( m' \) is defined as:

\[
\sigma_{h}^{i} (R, m') = \{(k', v') : h(K', R) = i, (k', v') \in m'\}
\]

Moreover, each partition \( p \) is partially ordered according to the keys, such that:

\[
\forall (k'_a, v'_a), (k'_b, v'_b) \in p, k'_a < k'_b \implies (k'_a, v'_a) \text{ precedes } (k'_b, v'_b) \text{ in } p
\]

However, no ordering is guaranteed “internally” to each given key. Then, the semantics of sort follows:

\[
\text{sort } h R m' = \{ \sigma_{h}^{i} (R, m') : \sigma_{h}^{i} (R, m') \neq \emptyset \}
\]

Finally, the reduce operator synthesizes the partitions on a per-key basis, according to a reduction kernel, respecting the ordering within each partition.

First, the reduce-by-key operator is defined, given a collection of key-value pairs, it produces a collection of synthesized collections, one for each key. Formally, given \( \oplus : K' \times \mathcal{P}(V') \rightarrow \mathcal{P}(V''') \) denoting the reduction kernel:

\[
\text{reduce-by-key } \oplus \ p = \{(k', \oplus \{v' : (k', v') \in p\}) : \exists (k', v') \in p\}
\]

Moreover, when applied to partitions (i.e., multi-sets partially ordered based on a per-key base), reduce-by-key produces a totally-ordered set, where all the values
with a given key are combined into a single key-value(s) pair and the ordering is kept among keys. Then, the reduce operator is defined as follows, where $m'_s$ is a multi-set of partitions and the big union operator respects the ordering:

$$\text{reduce} \oplus m'_s = \bigcup \{\text{reduce-by-key} \oplus p : p \in m'_s\}$$

**Parallel Execution**

A simple form of data parallelism can be exploited on the flat-map side, by partitioning the input collection into $n$ chunks and having $n$ executors process a chunk. In Dataflow terms, this corresponds to a graph with $n$ actors, each processing a token that represents a chunk. Each flat-map executor emits $R$ (i.e., the number of intermediate partitions) chunks, each containing the intermediate key-value pairs mapped to a given partition.

The intermediate chunks are processed by $R$ reduce executors. Each executor: 1. receives $n$ chunks (one from each flat-map executor); 2. merges the chunks into an intermediate partition and partially sorts it based on keys, as discussed above; 3. performs the reduction on a per-key basis. Finally, a downstream collector gathers $R$ tokens from the reduce executors and merge them into the final result.

A key aspect in the depicted parallelization is the shuffle phase, in which data is distributed between flat-map and reduce operators, according to an all-to-all communication schema. This poses severe challenges from the implementation standpoint.

**Run-time Support**

The most widespread implementation (i.e., Hadoop), is based in a Master-Workers approach, in which the master retains the control over the global state of the computation and informs the workers about the tasks to be executed.

A cornerstone of Hadoop is its distributed file system (HDFS), which is used to exchange data among workers, in particular upon shuffling. As a key feature, HDFS exposes the locality for stored data, thus enabling the principle of moving the computation towards the data, to minimize the communication. However, disk-based communication leads to performance problems when dealing with iterative computations, such as machine learning algorithms (Chu et al 2006).
Apache Spark

Apache Spark (Zaharia et al 2012) was proposed to overcome some limitations in Google MapReduce. Instead of a fixed processing schema, Spark allows datasets to be processed by means of arbitrarily composed primitives, referred as the application Directed Acyclic Graph (DAG). Moreover, instead of relying exclusively on disks for communicating data among the processing units, in-memory caching is exploited to boost the performance, in particular for iterative processing.

API and Semantics

Apache Spark uses a declarative processing style expressed as methods on objects representing collections. More precisely, Apache Spark implements batch programming with a set of operators, called transformations, that are uniformly applied to whole datasets called Resilient Distributed Datasets (RDD) (Zaharia et al 2012), which are immutable multisets. A Spark program can be characterized by the two kinds of operations applicable to RDDs: transformations and actions. Transformations are lazy functions (i.e., they do not compute their results right away) over collections—such as map, reduce, flatmap—that are uniformly applied to whole RDDs. Actions effectively trigger the DAG execution and they return a value to the user.

Listing 4 shows the source code for a simple Word Count application in the Java Spark API.

```java
JavaRDD<String> textFile=sc.textFile("hdfs://...");
JavaRDD<String> words =
    textFile.flatMap(new FlatMapFunction<String, String>() {
      @Override
      public Iterable<String> call(String s) {
        return Arrays.asList(s.split(" "));
      }
    });
JavaPairRDD<String, Integer> pairs =
    words.mapToPair(new PairFunction<String, String, Integer>() {
      @Override
      public Tuple2<String, Integer> call(String s) {
        return new Tuple2<String, Integer>(s, 1);
      }
    });
JavaPairRDD<String, Integer> counts =
    pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
      @Override
      public Integer call(Integer a, Integer b) {
        return a + b;
      }
    });
counts.saveAsTextFile("hdfs://...");
```

Fig. 4 A Java Word Count example in Spark from its examples suite.
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 according to the simple translation $\text{op}(a) = \left[ \text{op}(a_1), \text{op}(a_2), \ldots \right]$, 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.

When mapping a Spark program to a Semantic graph, tokens represent RDDs and DStreams for batch and stream processing respectively. Actors are operators—either transformations or actions—that transform data or return values (in-memory collection or files). Actors are activated only once in both batch and stream processing, since each collection (either RDD or DStreams) is represented by a single token.

**Parallel Execution and Runtime Support**

From the application DAG, Spark infers a parallel execution dataflow, in which many parallel instances of actors are created for each function and independent actors are grouped into *Stages*. Due to the Spark batch-oriented implementation, each stage that depends on some previous stages has to wait for their completion before execution, according to the classical Bulk Synchronous Parallelism (BSP) approach. Therefore, a computation proceeds in a series of global *supersteps*, each consisting of: 1) Concurrent computation, in which each actor processes its own partition; 2) Communication, where actors exchange data between themselves if necessary (the *shuffle* phase); 3) Barrier synchronization, where actors wait until all other actors have reached the same barrier.

Similarly to the MapReduce implementation, Spark’s execution model relies on the Master-Workers model: a cluster manager (e.g., YARN) manages resources and supervises the execution of the program. It manages application scheduling to worker nodes, which execute the application logic (the DAG) that has been serialized and sent by the master.

**Apache Flink**

*Apache Flink* (Carbone et al 2015) is similar to Spark, in particular from the API standpoint. However, Flink is based on *streaming* as a primary concept, rather than a mere linguistic extension on top of batch processing (as is in Spark). With the exception of iterative processing, stream parallelism is exploited to avoid expensive
synchronizations among successive phases, when executing both batch and stream programs.

API and Semantics

Apache Flink’s main focus is 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.

Listing 5 shows Flink’s source code for the simple Word Count application.

```java
public class WordCountExample {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
        DataSet<String> text = env.fromElements("Text...");
        DataSet<Tuple2<String, Integer>> wordCounts =
            text.flatMap(new LineSplitter()).
                groupBy(0).sum(1);
        wordCounts.print();
    }

    public static class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<>(word, 1));
            }
        }
    }
}
```

Fig. 5 A Java Word Count example in Flink from its examples suite.

Flink applications can be applied to semantic Dataflow graphs in the same way as for Spark, by treating DataSets and DataStreams as, respectively, RDDs and DStreams.

Parallel Execution and Runtime Support

Flink transforms a JobGraph into an ExecutionGraph, in which the JobVertex contains ExecutionVertexes (actors), one per parallel sub-task. A key difference com-
pared to the Spark execution graph is that, apart from iterative processing (that is still executed under BSP), there is no barrier among actors or vertexes: instead, there is effective pipelining.

Also Flink’s execution model relies on the Master-Workers model: a deployment has at least one job manager process that coordinates checkpointing and recovery, and that receives Flink jobs. The job manager also schedules work across the task manager processes (i.e., workers) which usually reside on separate machines and in turn execute the code.

**Apache Storm**

*Apache Storm* (Nasir et al 2015; Toshniwal et al 2014) is a framework targeting only stream processing. It is perhaps the first widely used large-scale stream processing framework in the open source world. Whereas Spark and Flink are based on a declarative data processing model—i.e., they provide as building blocks data collections and operations on those collections—Storm is instead based on a “topological” approach in that it provides an API to explicitly build graphs.

**API and Semantics**

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. Most of the logic of a computation goes into Bolts, such as functions, filters, streaming joins, or streaming aggregations. A Topology is the composition of Spouts and Bolts resulting in a network.

Storm uses *tuples* as its data model, that is, named lists of values of arbitrary type. Hence, Bolts are parametrized with per-tuple kernel code. It is also possible to define cyclic graphs by way of feedback channels connecting Bolts. Figure 6 show Storm’s source code for the simple Word Count application.

As for the mapping applications to Dataflow graphs, whereas in the previously described frameworks the graph is implicit and tokens represent whole datasets or streams, Storm is different: 1. The Dataflow graph is already explicit, as it is constructed using the provided API; 2. Each token represents a single stream item (tuple), and the actors, representing (macro) Dataflow operators, are activated each time a new token is available.
public class WordCountTopology {
    public static class SplitSentence extends ShellBolt implements IRichBolt {
        public SplitSentence() {
            super("python", "splitsentence.py");
        }
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }
    }
    public static class WordCount extends BaseBasicBolt {
        Map<String, Integer> counts = new HashMap<String, Integer>();
        @Override
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String word = tuple.getString(0);
            Integer count = counts.get(word);
            if (count == null) count = 0;
            count++;
            counts.put(word, count);
            collector.emit(new Values(word, count));
        }
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word", "count"));
        }
    }
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new RandomSentenceSpout(), 5);
        builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
        Config conf = new Config();
        conf.setDebug(true);
        conf.setNumWorkers(3);
        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
}

![Fig. 6 A Java Word Count example in Storm from its examples suite.](image)

**Parallel Execution and Runtime Support**

At execution level, each actor is replicated to increase the inter-actor parallelism and each group of replicas corresponds to the Bolt/Spout in the semantics Dataflow. Each of these actors represents independent data parallel tasks, on which pipeline
parallelism is exploited. Eventually, tasks are executed by a Master-Workers engine, as in the previous frameworks.

**Google Cloud Platform and Apache Beam**

Google Dataflow SDK (Akidau et al 2015) is part of the Google Cloud Platform. Google Dataflow supports a simplified pipeline development via Java and Python APIs in the Apache Beam SDK, which provides a set of windowing and session analysis primitives as well as an ecosystem of source and sink connectors. Apache Beam allows the user to create pipelines that are then executed by one of Beam’s supported distributed processing back-ends, which include, among others, Apache Flink, Apache Spark, and Google Cloud Dataflow, which are called runners.

**API and Semantics**

Apache Beam programming model is centered around the concept of Pipeline, that represents a data processing program consisting of a set of operations that can read a source of input data, transform that data, and write out the resulting output. A Pipeline can be linear but it can also branch and merge, thus making a Pipeline a DAG defined through conditionals, loops, and other common programming structures. A pipeline stage is a Transform, that accepts one or more PCollections (i.e., a possibly unbounded immutable collection, either ordered or not) as input, performs an operation on its elements, and produces one or more new PCollections as output. The ParDo is the core element-wise transform in Apache Beam, invoking a user-specified function on each of the elements of the input PCollection to produce zero or more output elements (flat-map semantics) collected into an output PCollection.

When mapping a Beam program into a semantic graph, tokens represent PCollections and actors are Transorms accepting PCollections in input and producing PCollections in output.

Figure 7 shows how to create a Word Count Pipeline.

**Parallel Execution and Runtime Support**

The bounded (or unbounded) nature of a PCollection also affects how data is processed. Bounded PCollections can be processed using batch jobs, that might read the entire data set once and perform processing as a finite job. Unbounded PCollections must be processed using streaming jobs—as the entire collection will never
static class ExtractWordsFn extends DoFn<String, String> {
    private final Counter emptyLines =
        Metrics.counter(ExtractWordsFn.class, "emptyLines");
    private final Distribution lineLenDist =
        Metrics.distribution(ExtractWordsFn.class, "lineLenDistro");

    @ProcessElement
    public void processElement(ProcessContext c) {
        lineLenDist.update(c.element().length());
        if (c.element().trim().isEmpty())
            emptyLines.inc();

        // Split the line into words.
        String[] words = c.element().split(ExampleUtils.TOKENIZER_PATTERN);
        // Output each word encountered into the output PCollection.
        for (String word : words)
            if (!word.isEmpty())
                c.output(word);
    }
}

public static class FormatAsTextFn
    extends SimpleFunction<KV<String, Long>, String> {
    @Override
    public String apply(KV<String, Long> input) {
        return input.getKey() + ": " + input.getValue();
    }
}

public static class CountWords
    extends PTransform<PCollection<String>, PCollection<KV<String, Long>>> {
    @Override
    public PCollection<KV<String, Long>> expand(PCollection<String> lines) {
        // Convert lines of text into individual words.
        PCollection<String> words =
            lines.apply(ParDo.of(new ExtractWordsFn()));
        // Count the number of times each word occurs.
        PCollection<KV<String, Long>> wordCounts =
            words.apply(Count.<String>perElement());
        return wordCounts;
    }
}

public static void main(String[] args) {
    // options initialization...
    Pipeline p = Pipeline.create(options);

    p.apply("ReadLines", TextIO.read().from(options.getInputFile()))
        .apply(new CountWords())
        .apply(MapElements.via(new FormatAsTextFn()))
        .apply("WriteCounts", TextIO.write().to(options.getOutput()));

    p.run().waitUntilFinish();
}

Fig. 7 Java code fragment for a Word Count example in Apache Beam from its examples suite.
be available for processing at any one time—and bounded subcollections can be obtained through logical finite size windows.

As mentioned, Beam relies on the runner specified by the user. When executed, an entity called Beam Pipeline Runner (related to execution back-end) translates the data processing pipeline into the API compatible with the selected distributed processing back-end. Hence, it creates an execution graph from the Pipeline, including all the Transforms and processing functions. That graph is then executed using the appropriate distributed processing back-end, becoming an asynchronous job/process on that back-end, thus the final parallel execution graph is generated by the back-end.

The parallel execution dataflow is similar to the one in Spark and Flink, and parallelism is expressed in terms of data parallelism in Transforms (e.g., ParDo function) and inter-actor parallelism on independent Transforms. In Beam’s nomenclature, this graph is called the Execution Graph. Similarly to Flink, pipeline parallelism is exploited among successive actors.

**Future Direction for Research**

In this chapter, some of the most common tools for analytics and data management were presented. One common drawback of all of them is that their operators are not polymorphic with respect to the data model (e.g., stream, batch). This means analytics pipelines should be either re-designed or simulated to match a different data model, as in the lambda or kappa architectures, respectively. This is often hardly acceptable either in term of effort or performance.

The PiCo framework has been recently proposed to overcome this problem. 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.). Being PiCo designed as a C++11 header-only library, it can be easily ported in any general-purpose or specialised device supporting vanilla C++ run-time, including any low-power device in the edge of computing. Preliminary results show that PiCo can attain equal or better performances in terms of execution times and hugely improve memory utilization when compared to Spark and Flink in both batch and stream processing (Misale 2017; Misale et al 2017b).
References


