λ-blocks: Data Processing with Topologies of Blocks

Matthieu Caneill
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
F-38000, Grenoble, France
caneill@imag.fr

Noël De Palma
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
F-38000, Grenoble, France
depalma@imag.fr

***Abstract*—We present and evaluate λ-blocks[1], a novel framework to write data processing programs in a descriptive manner. The main idea behind this framework is to separate the semantics of a program from its implementation. For that purpose, we define a data schema, able to describe, parameterize, compose, and link together blocks of code, storing a directed graph which represents the data transformations. Along this data schema lies an execution engine, able to read such a program, give feedback on potential errors, and finally execute it. In our reference implementation, a computation graph is described in YAML, linking together vertices of Python code blocks defined in separate libraries.**

The advantages of this approach are manyfold: faster, less error-prone programming; reuse of code blocks; computation graph manipulations; mixing of different specialized libraries; and finally middleware for potential front-ends (such as graphical interfaces) and back-ends (other execution engines). The main goal of λ-blocks is to bring complex data processing computations to non-specialists, by providing a simple abstraction over large-scale data processing systems.

Our contributions lie within a description of the schema, and an analysis of the reference execution engine. For that purpose we describe λ-blocks' internals and its main abstractions (blocks and topologies), and evaluate the framework performances. We measured the framework overhead to have a maximum value of 50 ms, a negligible amount compared to the average duration of data processing jobs.

I. Introduction

Within many frameworks and systems, data analysis can be summed up to a set of high-level operations: connect to a data store; fetch, clean and transform data; save the obtained result. Data is flowing from one operator to another, and the program can be easily represented with a directed graph, where vertices are operators and edges connect them together. For example, Apache Storm [1] allows to explicitly define such a graph when linking together its agents (spouts and bolts), and Apache Spark [2] automatically builds a lineage graph inferred from the successive methods called on its data structures (resilient distributed datasets). Many operations have been standardized in the fields of relational algebra or functional programming: *map*, *reduce*, *filter*, etc. Specialized libraries apply these operations to different data containers, sometimes on distributed clusters of machines, with different levels of optimization.

[1] λ-blocks is available under the Apache software license at https://github.com/lambdablocks/lambdablocks.

We argue these programs can be written in a higher-level fashion. By writing one or more of these operations in a "code block", we abstract out the functional code of this block, in the same manner as a library function. Having inputs and outputs, a block can then be a vertex of an oriented graph, which can be reused in different computations.

We propose to write such a graph in a descriptive fashion, rather than programmatic. Using for example YAML, a data serialization format particularly easy to read and parse, we can describe the vertices (linked to code blocks with unique names) and their edges (an edge exists when one block's output is another block's input). Moreover, a graph can itself be a sub-graph of another graph, leveraging code-reuse one step further. Some advantages of this approach include:

- strict separation of low-level data operations and high-level data processing programs;
- direct manipulation of the computation graph, for optimization, instrumentation, etc;
- reusability of code, with blocks being used in many programs, and computation graphs being composed of other graphs;
- easier reading, understanding, sharing, evolution and maintenance of a data processing program;
- seamless mixing of different frameworks and libraries together;
- as a middleware layer, room for front-ends such as graph visualization tools, and back-ends such as more optimized execution engines.

λ-blocks aims to bring framework-agnostic dataflow programming to large-scale data processing, without losing any of the benefits provided by specialized and well-optimized libraries implementing data operations.

The rest of this paper is organized as follows: we first dive into λ-blocks' design, its format for topologies and blocks, and its execution engine. We then present examples of processing graph manipulations, before evaluating the framework's performances. We finally introduce some related work and conclude.

978-1-5386-7232-7/18/$31.00 ©2018 IEEE
DOI 10.1109/BigDataCongress.2018.00009

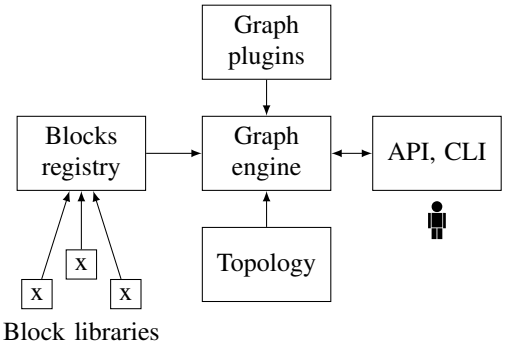


Figure 1: System architecture

II. λ-BLOCKS

A. Terminology

We describe here the different abstractions used in λ-blocks. These objects need to exist in any engine implementing the system:

- **Block**: a standalone piece of code, able to provide information about its behavior, either using introspection (with languages supporting it) or embedded declarations (e.g. through decorators or class attributes). A block needs to describe at least its ports and arguments (explained below).
- **Component**: an *instance* of a block, i.e. a block with its arguments, ready to run its functional code.
- **Port**: a named input or output of a block. The inputs are the data provided to the block as arguments, and the outputs are its results.
- **Argument**: a runtime option of a block, such as a configuration value. It is different from an input port, because it is not meant to carry flowing data, but rather a variable to parameterize the block, initialized in the topology.
- **Registry**: a catalog of blocks, providing their functional code along with their metadata (a list of key-value pairs used to classify blocks) and documentation.
- **Topology**: a computation graph, i.e. a high-level representation of a data processing program, defining components and linking them together as a DAG. Sometimes simply called graph.
- **Sub-topology**: a topology used as a component of another topology (i.e. when using a graph as a vertex of a bigger graph). Sometimes called sub-graph.

B. Architecture

Figure 1 shows the architecture of the system. Its different components are as follows:

- The graph engine is the main controller, it is responsible for parsing topologies, matching the vertices with code blocks, building the corresponding graph, running different checks against it, and finally executing it.
- Graph plugins can add functionalities, such as graph manipulation, instrumentation, etc. They are plugged to the graph engine.
- The blocks registry aggregates the code blocks defined in different blocks libraries, and extracts their metadata, either through introspection or through decorators or tags.
- A topology is the main input of the system, defined in one or more files.
- Finally, the system can be driven by an API and a CLI, and more front-ends can be plugged, for instance graphical interfaces.

We kept the architecture modular, so that it is easily extendable, and the different components can be replaced independently; this can be useful for instance to extend the supported description and programming languages (beyond YAML and Python).

C. Topologies format

The main design goals of the topology schema are simplicity and extensibility. It is meant to be easily written by hand, even by non-programmers, who would simply need to know the high-level concepts of data transformations and the YAML syntax.

YAML is a data-serialization language, focusing on readability. It can easily define lists and associative arrays, and supports typing. We chose it for the simplicity of reading and writing data with it, and because it is fast to parse.

There are two types of objects in a topology schema, *blocks* and *sub-topologies*, described below.

1) Defining and linking components: A topology consists of two YAML sections: the first one is a simple dictionary, which permits to assign a name and various metadata to the topology. It is useful when the number of maintained topologies grows within an organization, and it permits to easily retrieve them through a search engine for example. No key is mandatory in this first section, except the `name` when this topology is to be composed with other ones, as it needs a unique identifier.

The second section lists the different components and links them together. Some keys are defined in the reference implementation, and it is easy to add new ones to further customize the topology. These keys are:

- `block`: the name of the block (from the blocks registry) that is to be used;
- `name`: a unique name for this component;
- `args`: optional, it allows to give arguments to the block, to customize its behavior;
- `inputs`: absent for entry-level blocks (the data sources), it permits to link components together.

Both `args` and `inputs` are defined with a dictionary, because they are always explicitly named. A block can have zero or more inputs, and zero or more outputs. At the topology level, only the inputs are defined; its outputs are

```
---
name: count_users
description: Count number of system users
---
- block: readfile
  name: my_readfile
  args:
    filename: /etc/passwd

- block: count
  name: my_count
  inputs:
    data: my_readfile.result
```

Listing 1: A simple topology

inferred from the other blocks consuming them, and can sometimes remain unused (if no other block subscribes to them).

Listing 1 shows a simple topology, which counts the users of a Linux-based system (note it is incorrect since it will also count daemons and other system users, but this is out of the scope of the example). The first block will read a file line by line, and it knows which file to open through the `args.filename` value. The second block will simply count the length of the data structure it receives: it has one input, named `data`, which is linked to the output `result` of the named block `my_readfile`. This block produces another result, accessible through `my_count.result`, which could be displayed on a console, saved to a file, or used as an input of other blocks.

2) Encapsulating other topologies: The second type of component which can be defined in a YAML topology is a topology itself, encapsulated and linked to blocks or other sub-topologies. This poses a few challenges, mainly to keep the outer links simple to define.

An example is shown in Listing 2, along with its graph representation in Figure 2. On the left column, the main topology is defined, it instantiates a sub-topology block, `count_pb`, which is linked to two blocks, `readfile` and `print`. The `bind_in` dictionary permits to give inputs to this sub-topology, while `bind_out` permits to use some of its outputs as inputs to other blocks. The sub-topology is displayed on the right column.

Any block output can be used as a sub-topology input, and once they are defined, they are accessible in the encapsulated topology through the special dictionary `$inputs`. We keep the $ sign as the only reserved symbol, which could be used in the future for different matters, for example to give command-line parameters to a topology.

Similar to the bound inputs, any output of any block of the sub-topology can be linked to a block. There is no need to use a special dictionary for this purpose, since outputs are never explicitly declared. In this example, the value `count_pb.result` means the output `result` of the block `count` of the encapsulated topology `count_pb`.

```
---
name: foo_errors
---
- block: readfile
  name: readfile
  args:
    filename: foo.log

- topology: count_pb
  name: count_pb
  bind_in:
    data: readfile.result
  bind_out:
    result: count.result

- block: print
  name: print
  inputs:
    data: count_pb.result
```

```
---
name: count_pb
---
- block: filter
  name: filter
  args:
    contains: error
  inputs:
    data: $inputs.data

- block: count
  name: count
  inputs:
    data: filter.result
```

Listing 2: Encapsulation example. Left: the main topology; right: the encapsulated topology. Reports the number of errors found in a file `foo.log`.

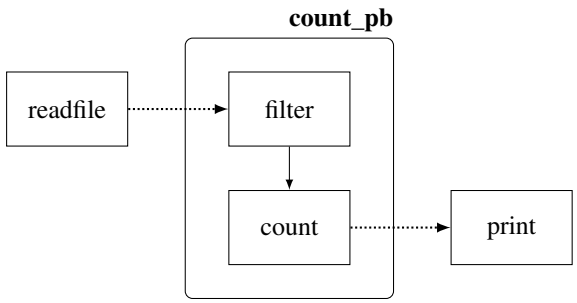


Figure 2: The DAG associated to the program in Listing 2. The *bind_in* and *bind_out* links for the sub-graph are dotted.

The result of this is a program filtering errors in a log file, counting them, and displaying the sum. It makes a great use of encapsulation, because the sub-program taking care of filtering and counting could be used in other topologies, for example with other log files as bound input, or with a block saving the result in a database as bound output.

D. Block internals

Blocks are the individual components of the topologies. They are independent and reusable: they know nothing about a data processing program, except their inputs and outputs. Most of them only take care of transforming data, and hence don't have side effects, in a purely functional manner. Some blocks read data from stores (they are the entry points of the computation graph), and some save back their results on storage. There is no restriction about the programming language used to write the blocks, as long as they can be called from the engine manipulating them. We chose Python for the reference implementation for a few reasons:

```python
@block(engine='localpython')
def take(n: int=0):
    """
    Given a list of integers, returns the n
    first items.
    """
    def inner(data: List[int])->ReturnType[List[int]]:
        assert n <= len(data)
        return ReturnEntry(result=data[:n])
    return inner
```

Listing 3: A typical block structure

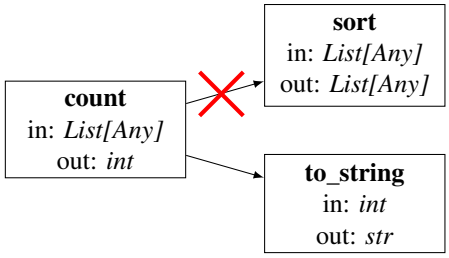


Figure 3: Type checking

- Simplicity: it is a design goal of λ-blocks to be as simple as possible, and Python has been known for being very accessible to novice programmers.
- Variety of libraries: since blocks can wrap any library function, Python is a good choice for combining distributed computing (for example through pyspark, the Python package to interact with Spark), machine learning (MlLib, scikit-learn), plotting (matplotlib), etc.
- Introspection: it is straightforward to inspect functions in Python, hence the engine can infer a lot of metadata about a block (its arguments, inputs, outputs, and documentation) without them being declared explicitly. Any other metadata can be added with function decorators.

An example is shown in Listing 3. A block named `take` is registered through the `@block` decorator, which takes any pair of key/value for tagging it, for example to categorize it. We then create a closure: the outer function takes the block's arguments, while the inner function takes the block's inputs. We use Python's type annotation capabilities to give types to the arguments, inputs and outputs. The special `ReturnType` and `ReturnEntry` give us the ability to properly define the block's outputs, to overcome some limitations of the dynamic manipulation of Python's typing annotations. This way, the arguments, inputs and outputs can all be documented and verified.

What happens in the inner function is the responsibility of the block developer, and can be anything Python can do, such as direct data manipulation, library function wrapping, or input/output (to retrieve and store data).

E. Execution engine

The execution engine is the glue between the topologies and the Python blocks of code. Upon initialization, it will parse a topology, build the associated DAG (recursively when sub-topologies are involved), and associate each vertex with a named block of code. It can then run some DAG manipulations, do all the necessary checks, and execute the graph, giving each component its inputs after they have been computed.

The execution engine must be fast, to reduce the overhead of the system as much as possible, and also easy to extend, to leave the possibility of adding graph manipulations. For the latter, atop the internal API to manipulate edges, vertices, and the engine itself, it provides a plugin system, which makes it easy to register hooks at the different steps of the graph execution. When these hooks are called, they receive relevant parameters, such as the current value of the flowing data for a certain block. Some possible manipulations are described in the next section, showing both the use of the internal API and the plugin system.

In the reference implementation, the engine is single-threaded. However, each component can easily leverage parallelization by spawning multiple threads. While this is not optimal for building a proper distributed system, it is not λ-blocks' role: distributed data processing frameworks such as Apache Spark do it better. λ-blocks is meant to wrap their instructions in order to combine their benefits with its own.

III. Topology abstraction

As stated earlier, having a high-level representation of the processing topology can bring many benefits, among them the possibility of observing and optimizing a data processing program. We describe as an example the use of type-checking to perform verifications between connected blocks, and dive into the plugin interface λ-blocks provides to let developers implement their own topology manipulations.

A. Type checking

Python does not benefit from compile-time type safety. However, it supports type annotations for variables and functions, and these annotations include base types as well as more complex ones, such as generic lists and dictionaries, unions, callables, etc. Type checking can only happen statically, with the Mypy [3] static analyzer. The links between the different blocks being computed dynamically (from their YAML description), we implemented a type checker, which runs right after the DAG construction. The types of every vertex input and output have already been introspected, hence it is enough to check that the types of both ends of an edge are compatible.

An example of type checking is shown in Figure 3. The block `count` takes a list of elements, and returns the length of the list, as an integer. This is fine for the block `to_string`, which takes an integer as an input, however this doesn't make sense for the block `sort`, which takes

a list as an input. The type checker, when verifying the edge between the blocks `count` and `sort`, will see two incompatible types at its extremities, and will raise an error.

This process was easily implemented thanks to the high-level DAG manipulation features λ-blocks provides: iterating through vertices and edges, accessing blocks' details, and accessing the registry's objects (functions, input types, etc). This feature is useful on its own, but can also be leveraged when writing a graphical interface: an edge could simply not be created between two vertices if their types were not compatible. This reduces potential errors while writing data processing programs, giving an immediate feedback to the user.

B. Plugins

λ-blocks' topologies can be enhanced by plugins, pieces of code which can register hooks to be run at different stages of the execution. They access the relevant datastructures (topologies and individual blocks) and can modify them in place.

Plugins can be categorized according to when their hooks are executed: passive plugins run code before and after the topology execution (for example to perform verifications, offline optimizations, etc), while active plugins run code during the execution (for example for online optimizations or measurements).

As an example, we describe how to implement a plugin for instrumenting topologies, i.e. measuring the time taken by each of its components. We define three hooks:

- before block execution: stores a timestamp associated to this block;
- after block execution: computes the time it took for the block to execute, using the previous timestamp;
- after graph execution: sorts and displays all the recorded durations.

Listing 4 shows an excerpt of the instrumentation plugin implementation. With less than 20 lines of code, this plugin is able to determine the slowest blocks of a topology, to give developers insights on how to optimize their programs.

This example shows the easiness with which one can implement plugins to manipulate topologies, which brings additional benefits to the idea of separating code from processing chains.

```
by_block = {} # timing by block: begin, duration

@before_block_execution
def store_begin_time(block):
    name = block.fields['name']
    by_block[name]['begin'] = time.time()

@after_block_execution
def store_end_time(block, results):
    name = block.fields['name']
    by_block[name]['duration'] = \
      time.time() - by_block[name]['begin']

@after_graph_execution
def show_times(results):
    longest_first = sorted(by_block, reverse=True)
    for blockname in longest_first:
        print('{}\t{}'.format(
            blockname,
            by_block[blockname]['duration']))
```

Listing 4: Instrumentation plugin

IV. Evaluation

A. Performances

The first metric we want to calculate is the overhead of using λ-blocks' engine, compared to a regular Python program. We run 3 different programs, and compare their execution times on different setups:

- with λ-blocks;
- with λ-blocks and two plugins: *debug* (which displays the intermediary results computed by every block) and *instrumentation* (described in Section III-B);
- without λ-blocks, writing the equivalent code in a regular Python fashion.

The three programs we run show different patterns of latency and complexity:

- Wordcount on trending Twitter hashtags: we run the example shown in Listing 5, which extracts hashtags from the Twitter API, and groups and counts them in a wordcount sub-topology (a general-purpose topology that defines a group-by/count/sort/head processing chain for any input). This program has a non-negligible network overhead, since it needs to wait for the http queries to complete before continuing. It is single-threaded.
- Wordcount on a local file (without network queries), over a Wikipedia dataset [4] which we trimmed to consist of 10 million words. This dataset is an HTML dump of the English version of Wikipedia. The program is very similar to the previous one (blocking on disk instead of network) and is single-threaded.
- PageRank on an Apache Spark cluster, over a dataset of internal Wikipedia hyperlinks [5]. The DAG has two entry points (the file containing the links, and the one containing the page names, both stored in HDFS), and the blocks use various Spark functions. It is run on a bare-metal Spark server, and is an example of how blocks can be simple wrappers around other data processing frameworks.

To obtain more precise results, we run the Twitter Wordcount 10 times for each setup (with the program latency, that's the limit to not reach the API rate limits), the Wikipedia Wordcount 1000 times, and the Spark PageRank

```
---
name: twitter-wordcount
description: Extract the most used hashtags on the
             recent AFP timeline.
---
- block: twitter_search
  name: twitter_search
  args:
    query: "from:afp"
    client_key: xxx
    client_secret: xxx
    resource_owner_key: xxx
    resource_owner_secret: xxx

- block: flatMap
  name: extract_hashtags
  inputs:
    data: twitter_search.result
  args:
    func: "lambda x: [y['text'] for y in \
             x['entities']['hashtags']]"

- topology: topology-wordcount
  name: wordcount
  bind_in:
    data: extract_hashtags.result
  bind_out:
    result: head.result

- block: show_console
  name: show_console
  inputs:
    data: wordcount.result
```

Listing 5: Wordcount over the most recent hashtags from the press agency AFP, using a sub-graph.

10 times. We kept the average of the obtained values as our reference.

The times are measured with `/usr/bin/time -p`: *real* is the time taken by the program to complete; *user* is the CPU time consumed in user mode by the program, and *sys* is the CPU time consumed in kernel mode. If *user* and *sys* don't add up to *real*, it means the program was blocked during execution, generally waiting for disk or network.

Figure 4 shows the results obtained. The first program, in Figure 4a, confirms there is indeed a network overhead, during which the program is waiting (in the three cases), and is not consuming CPU cycles. More importantly, by substracting times, we measure the overhead of using λ-blocks: about 50 ms per run (we see in Section IV-B how it can be reduced further). The last interesting point is the negligible difference between λ-blocks with and without plugins: inspecting the graph vertices and instrumenting their computation times comes almost for free (the difference is smaller than the standard deviation of multiple runs in the same setup).

Figure 4b shows almost the same execution times for the three setups, and the first one (with λ-blocks) is even faster. This is not supposed to be the case, because it executes more code by design, but comes from the non-determinism of disk (and kernel cache) input/output: the speed varies with time, hence the imprecision of the calculation. The key point here is that using λ-blocks doesn't add any significant overhead if the job runs over a few seconds.

Finally, Figure 4c depicts the execution times for the PageRank computed with Spark. The first thing to notice is the low values of *user* and *sys* times, too low to be visible on the plot. This time it is not due to the majority of the program waiting for IO, but rather because the programs are communicated to and executed by a Spark daemon. Hence the CPU times are not seen by `/usr/bin/time`. However the *real* times are correct, and we can use them to compare the setups. Similar to the previous experiments, using λ-blocks doesn't add any significant overhead; and for such a job duration (about 14 minutes), it is negligible. Hence λ-blocks can easily drive a Spark program at a low cost.

B. Engine instrumentation

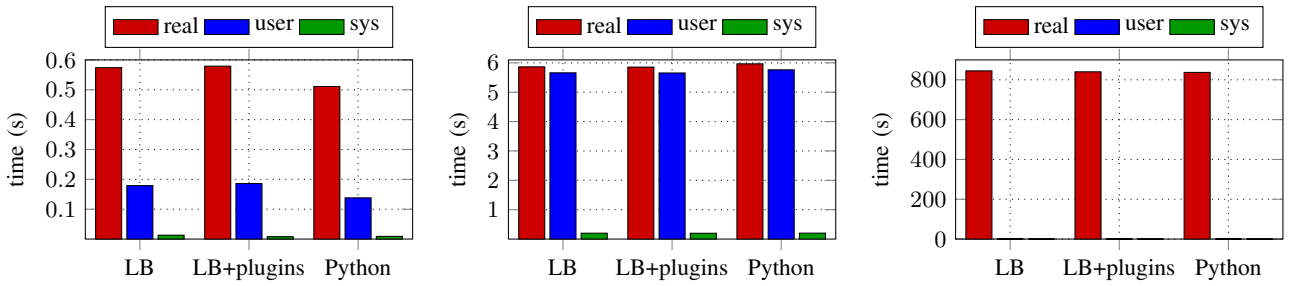
In order to further reduce the overhead added by the use of λ-blocks' engine compared to writing regular Python programs, we instrument the framework when running the Wikipedia Wordcount described above. We used a smaller input file in order to have a total execution time comparable to the measured overheads. We instrument three different setups, only changing the command-line parameters of λ-blocks:

- Loading all the block modules, and two plugins;
- Loading only one block module, and two plugins;
- Loading only one block module, without any plugin.

Figure 5 shows the results we obtained, with the different steps followed by λ-blocks : (1) Python startup, modules import and arguments parsing; (2) Blocks registry creation, block modules import; (3) Plugin import; (4) YAML parsing and graph creation; (5) Graph checks; (6) Graph execution. We note interesting results:

- Importing and executing plugins doesn't add any visible overhead, which confirms the results described in Section IV-A.
- Importing all the available blocks in the built-in block modules is very costly: between 250 and 300 ms. Python's import mechanisms are known to being slow [6]. This is why we only imported selected block modules in the previous section, to obtain a smaller framework overhead.
- Building the computation graph, and running checks against it (correct YAML, type checking, absence of loops, etc), is very fast; this is encouraging to develop more graph manipulation plugins.

Overall, the best optimization we found is to avoid importing block modules if they are not to be used. We target our future work to develop a λ-blocks daemon, in order to load modules only once and execute many computation graphs on demand.



(a) Twitter hashtags Wordcount (b) Wikipedia file Wordcount (c) Wikipedia hyperlinks PageRank

Figure 4: Time taken to process different programs, with and without λ-blocks.

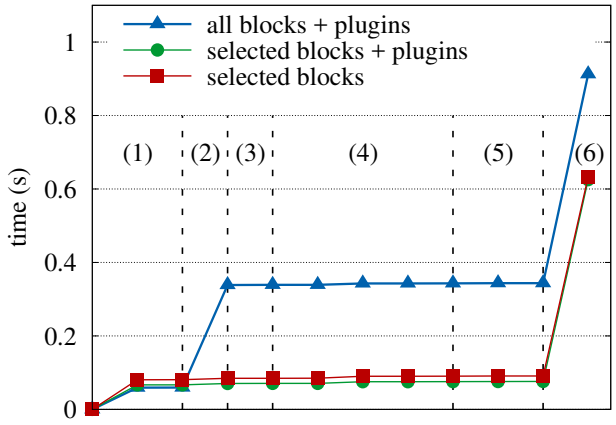


Figure 5: Instrumentation of a Wordcount program running under different setups.

V. Related work

Blocks-based programming has gained a lot of attention recently [7]. λ-blocks shares similar ideas, for example combining chunks of embedded code to create larger programs. However, block-based graphical interfaces are not oriented towards large scale data processing, and hence do not benefit from distributed libraries such as Apache Spark. We plan to explore some of their innovative features such as *recognition over recall* [7], immediate feedback, and impossibility to link blocks that don't make sense together. We believe it is a path towards bringing data processing and analysis to non-programmers.

Graphs from configuration is not a novel concept either. Pyleus [8] and later Storm Flux [9] brought configuration-based topologies to the Apache Storm [1] framework, for stream-processing. They both use a YAML format to define topologies, and inspired λ-blocks. However, they are limited to control Storm's own objects, spouts and bolts, which are meant to process streaming data.

Dataflow programming with pipelines has been implemented in numerous frameworks. For machine-learning applications for example, scikit-learn [10] and Apache Spark [11] have a built-in concept of Machine Learning Pipelines, where different data processors are defined and linked with each other. However, the DAGs are created programmatically in their respective library languages, and they are limited to the components of their frameworks.

The Orange framework [12] features a collection of widgets, linkable with each other, to execute and visualize machine learning algorithms. Like λ-blocks, it has a programming interface to implement new widgets in Python.

StreamPipes [13] is a framework for building and executing data stream pipelines, oriented towards distributed real-time processing of data, between sources and sinks. Some of their ideas are similar to those of λ-blocks (type checking (and other verifications) between operators, independent and self-contained blocks of code), but the framework differs with regard to some design choices: stress towards wrapping external computing engines, formalization of message passing between operators with data serialization formats (no possibility to use direct memory transfers), and RDF as the description and configuration language. Finally, it is oriented towards real-time data, whereas λ-blocks focuses on offline analysis.

Cascading [14] is a layer on top of Apache Hadoop. It permits to programmatically describe MapReduce jobs, by linking components together (sources, pipes, and sinks) in any JVM-based language. It has some similarities with λ-blocks, but also different goals: restricted to Hadoop and Flink, no graph manipulation, no configuration-oriented description of jobs.

KeystoneML [15] is a framework written in Scala, leveraging the use of high-level operators to build machine learning pipelines. Like the other introduced frameworks, it has different goals than λ-blocks', but shares the dataflow design and the reuse of components.

Other related tools [16], [17], [18], [19], [20] exist, but to the best of our knowledge, none implements all the features of λ-blocks, in particular the DAG specifications and the high-level graph manipulation abstractions.

VI. Conclusion and future work

We presented λ-blocks, a system which permits to define execution graphs for large scale data processing, combining blocks of code with high-level manipulable directed acyclic graphs. We described a reference implementation of λ-blocks, which uses Python as the language for blocks and YAML as the data-serialization format for topologies. We explained the design choices, the system internals, and evaluated the framework. λ-blocks has a very small overhead with regards to a system which doesn't use explicit computation graphs.

As future work, we want to explore how we can further reason about these graphs. λ-blocks allows to work with a high-level DAG, which opens opportunities for graph complexity analysis, serialization methods comparison, caching optimizations, automatic choosing of data processing libraries, in-depth monitoring, and verification of programs' semantics. Another axis we want to explore is data streaming, and how we can simply and efficiently implement continuous queries on online data; as well as for example implementing triggers with blocks, which could be used for many automation tasks beyond the scope of data analysis. Another open problem is the representation of lambda functions: as of today, we used Python's notation, because the execution engine is written in Python. We plan to further explore the possibilities to stay language-agnostic for this issue. Finally, we want to search new ways to mirror library APIs, in order to simplify the writing and maintenance of block collections.

Acknowledgment

This work has been supported by the Smart Support Center FEDER EU project, and the Project Pia FSN Hydda. We also thank all the people who have contributed design ideas and helped with the development of λ-blocks.

References

[1] 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*, ser. SIGMOD '14. ACM, 2014, pp. 147–156.

[2] 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*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[3] Jukka Lehtosalo et al., "Mypy," http://mypy-lang.org/.

[4] "Puma benchmarks and dataset downloads," https://engineering.purdue.edu/~puma/datasets.htm.

[5] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 555–564. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098069

[6] "PythonSpeed/PerformanceTips - Python Wiki," https://wiki.python.org/moin/PythonSpeed/PerformanceTips#Import_Statement_Overhead.

[7] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Commun. ACM*, vol. 60, no. 6, pp. 72–80, May 2017. [Online]. Available: http://doi.acm.org/10.1145/3015455

[8] YelpArchive, "Pyleus," https://github.com/YelpArchive/pyleus, 2016.

[9] The Apache Storm developers, "Flux," http://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html, 2017.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[11] The Apache Spark developers, "ML Pipelines," https://spark.apache.org/docs/latest/ml-pipeline.html, 2017.

[12] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html

[13] D. Riemer, F. Kaulfersch, R. Hutmacher, and L. Stojanovic, "Streampipes: solving the challenge with semantic stream processing pipelines," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. ACM, 2015, pp. 330–331.

[14] "Cascading — Application Platform for Enterprise Big Data," http://www.cascading.org/.

[15] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "Keystoneml: Optimizing pipelines for large-scale advanced analytics," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 535–546.

[16] "Pipeline.io," http://pipeline.io/.

[17] "Xplenty," https://www.xplenty.com/is/.

[18] "Blaze," http://blaze.pydata.org/.

[19] "Pipes.digital," https://www.pipes.digital/.

[20] "Flowhub and Noflojs," https://flowhub.io/.