```
%install_ext https://raw.github.com/cjdrake/ipython-magic/master/gvmagic.py
%load_ext gvmagic
# %matplotlib inline
import numpy as np
import pandas as pd
from yamal.defaults import defaults
from yamal.observers.graph_observer import GraphObserver
from yamal.observers.graphviz_graph import GraphvizGraph
from yamal.subprocess import SubprocessYamalExecutor
import matplotlib.pyplot as plt
import warnings

# warnings.filterwarnings("ignore")
# warnings.resetwarnings()

ggr = GraphvizGraph(mode="functions")
defaults["observers"] = [GraphObserver(ggr, output_node=False, compact_reduc
er=True)]

# defaults["executor_factory"] = lambda loop: SubprocessYamalExecutor(2, loo
p=loop)

def get_data():
    return pd.DataFrame({"LOCATION": range(10), "SALES": 0})

def get_larger_data():
    rc = 10000
    return pd.DataFrame({"c{}".format(i): np.random.randn(rc) for i in range
(100)})

def get_chunk(i):
    return get_data()

def get_chunked_data(chunks):
    for _ in range(chunks):
        yield get_larger_data()

def compute(df, parameter=None):
    return df

def compute_2(df):
    return df

def compute_3(df):
    return df

def create_plot(df):
    return df

def create_plot_2(df):
    return df

def create_report(df):
    return None
```

```
def write_report(df):
    return None

def clear_graph(mode="functions"):
    global ggr
    ggr = GraphvizGraph(mode=mode)
    defaults["observers"] = [GraphObserver(ggr, output_node=False, compact_r
educer=True, user_label=True)]

def show_graph(label_dict=None, mode="functions"):
    global ggr
    if label_dict is not None:
        for node in ggr.node_array:
            node_attr = ggr.node_attrs[node]
            label = node_attr.get("label", str(node))
            if label in label_dict:
                node_attr["label"] = label_dict[label]
    %dotstr ggr.dot_file()

    ggr = GraphvizGraph(mode=mode)
    defaults["observers"] = [GraphObserver(ggr, output_node=False, compact_r
educer=True, user_label=True)]
```

```
Installed gvmagic.py. To use it, type:
  %load_ext gvmagic
The gvmagic extension is already loaded. To reload it, use:
  %reload_ext gvmagic
```

Plumbing in Python: Pipelines for Data Science Applications

Thomas Reineking

What is Blue Yonder?

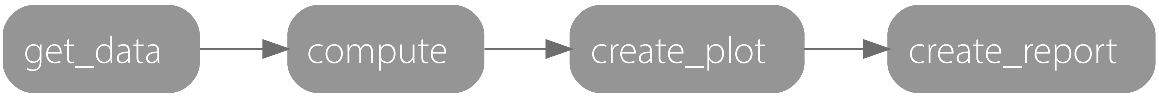
- Data science company
- Predictive applications for customers in retail
    - Replenishment optimization
    - Price optimization
- More than 500 billion automated decisions per month

Examples of what we do for our customers

- Machine learning
- Data analysis
- Reporting

Series of processing steps where the output of one step is the input for the next

```
from yamal import run_pipeline
pipeline = [
    get_data,
    compute,
    create_plot,
    create_report
]
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



What this talk is about

- Data flow library for creating pipelines
- Effect on how we develop software

Outline

- Part I: Pipelines
- Part II: Tools and technical details
- Part III: Effects on development

Part I: Pipelines

Example of processing some data

```
# get a lot of data:
data = get_data()

# do some fancy analysis/machine learning:
data = compute(data)

# create some plot:
plot = create_plot(data)

# create report:
report = create_report(plot)
```

Processing in pipelines

- Data is typically being processed in pipelines
- It would be nice to make this more explicit
    - Encourage developers to write code as pipelines
- This is why we developed **Yamal** at Blue Yonder

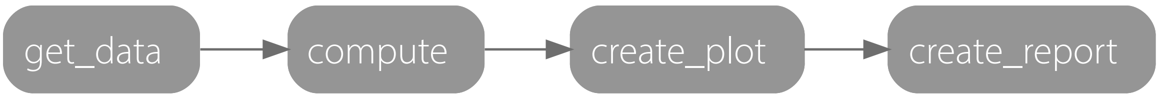
Simple pipeline example

```
data = get_data()
data = compute(data)
plot = create_plot(data)
report = create_report(plot)
```

```
pipeline = [
    get_data,
    compute,
    create_plot,
    create_report
]
```

```
from yamal import run_pipeline
run_pipeline(pipeline)
```

```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



Advantages of using pipelines

```
pipeline = [get_data, compute, create_plot, create_report]
run_pipeline(pipeline)
```

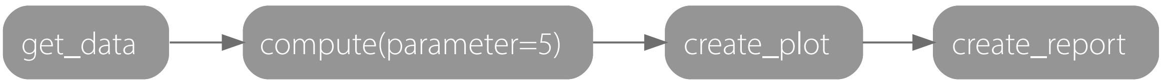
- Separation of declaration and execution
    - Code does not depend on execution backend
- Encourages functional programming style
- Pipelines are simply Python lists
    - Concatenation, slicing, ...
- Nice side-effect: Easy to do parallelization

Defining pipelines using Yamal

Binding arguments

```
from functools import partial

pipeline = [
    get_data,
    partial(compute, parameter=5),
    create_plot,
    create_report
]

run_pipeline(pipeline)
```

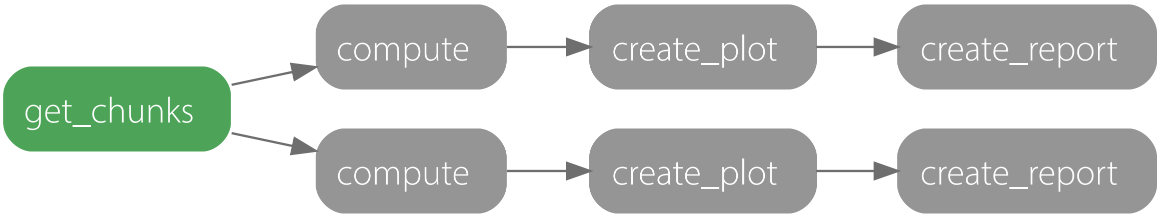
```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



Splitting data using generators

```
from yamal import splitter

def get_chunks():
    yield get_chunk(0)
    yield get_chunk(1)

pipeline = [
    splitter(get_chunks),
    compute,
    create_plot,
    create_report
]

run_pipeline(pipeline)
```

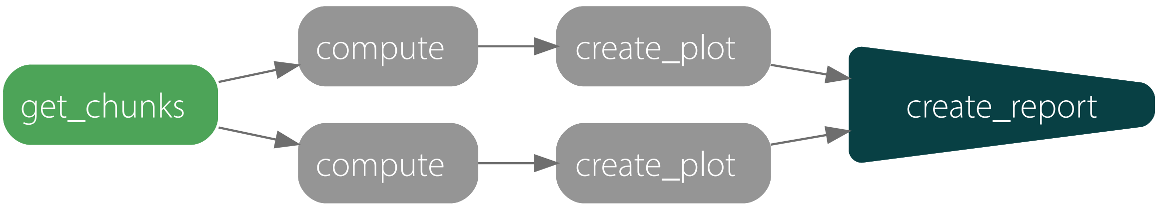
```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



Split-map-reduce

```
from yamal import reducer

def create_report(plots):
    pass  # create report from list of plots

pipeline = [
    splitter(get_chunks),
    compute,
    create_plot,
    reducer(create_report)
]

run_pipeline(pipeline)
```

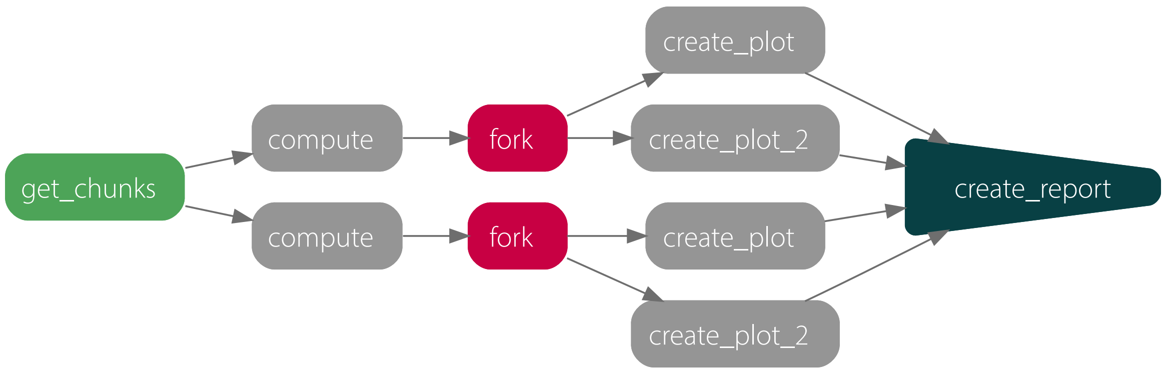
```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



Passing the same data to different functions

```
from yamal import fork

pipeline = [
    splitter(get_chunks),
    compute,
    fork([create_plot, create_plot_2]),
    reducer(create_report)
]

run_pipeline(pipeline)
```

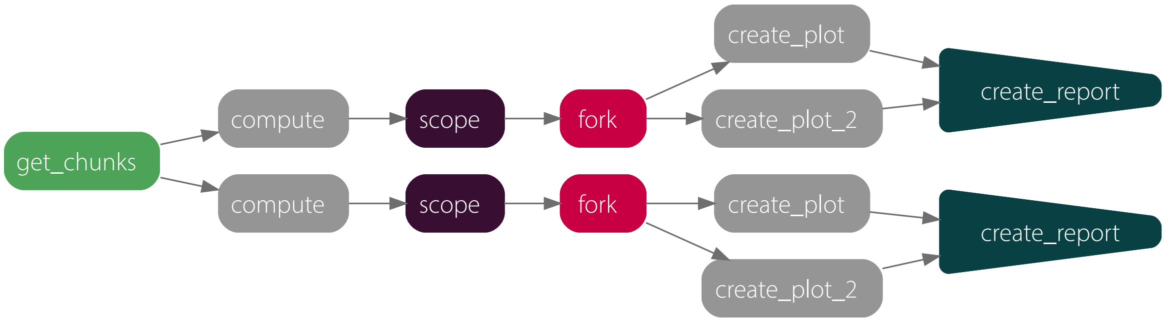
```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph({"distribute_to": "fork"})
```



Scopes for nested pipelines

```
from yamal import scope

pipeline = [
    splitter(get_chunks),
    compute,
    scope([
        fork([create_plot, create_plot_2]),
        reducer(create_report)
    ])
]

run_pipeline(pipeline)
```

```
clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph({"distribute_to": "fork"})
```

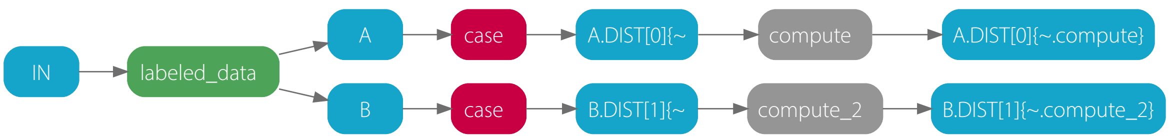


Labels and control flow

- Assign labels to data
- Pass labeled data to particular functions

```
from yamal import case, returns_label

def labeled_data():
    yield "A", get_chunk(0)
    yield "B", get_chunk(1)

pipeline = [
    returns_label(splitter(labeled_data)),
    case().\
    when(last_label_equals="A", pipeline=[compute]).\
    when(last_label_equals="B", pipeline=[compute_2])
]

run_pipeline(pipeline)
```

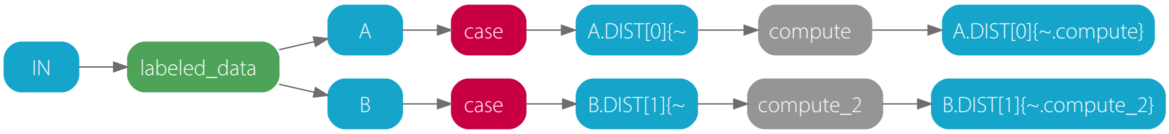
```
clear_graph(mode="all")
run_pipeline(pipeline, suppress_fusing=True)
show_graph({"distribute_to": "case"})
```



Labels and control flow

- Functions do not need to be aware of control flow
- Which function is called with which arguments is determined dynamically

```
clear_graph(mode="all")
run_pipeline(pipeline, suppress_fusing=True)
show_graph({"distribute_to": "case"})
```



How do pipeline functions look like?

```
def function(input):
    # fancy computation here
    return output
```

- Single argument (unless first in pipeline)
- Typically pure
  - Good for testability
- No dependency on Yamal
  - Does not need to know about pipeline

What did we gain?

- High re-usability (not by Yamal but induced by design conventions)
- Different execution backends can be used
- Building blocks and their interaction directly visible

Part II: Tools and technical details

How does Yamal work internally?

- Pipelines are lists of (annotated) Python functions
- Functions and data are pickled using dill
- Execution backends schedule jobs using asyncio (Trollius)

```
clear_graph(mode="all")
run_pipeline([get_data, compute], suppress_fusing=True)
show_graph()
```

Pipeline observers

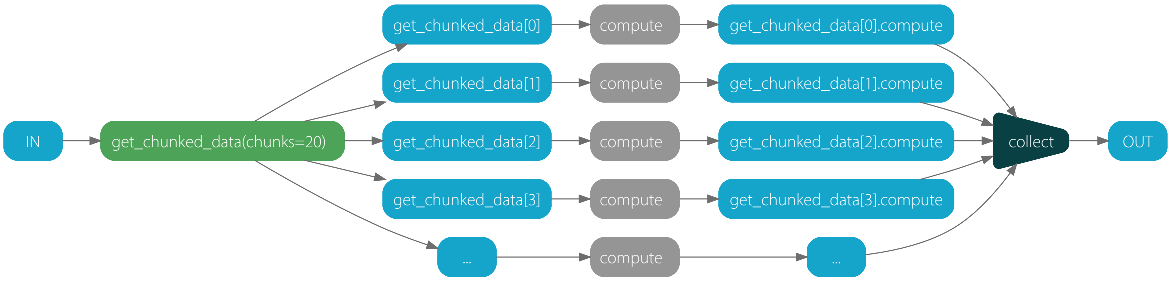
- Monitor pipeline execution in real-time
- Debugging
- Performance optimization

```
from yamal.pipeline_observer import PipelineObserver

observer = PipelineObserver()

run_pipeline(pipeline, observers=[observer])
```

Graph observer

```
from yamal.observers import GraphObserver, GraphvizGraph

pipeline = [
    splitter(partial(get_chunked_data, chunks=20)),
    compute
]

graph = GraphvizGraph()
observer = GraphObserver(graph, max_degree=4)
run_pipeline(pipeline, observers=[observer])
```

```
%dotstr graph.dot_file()
```

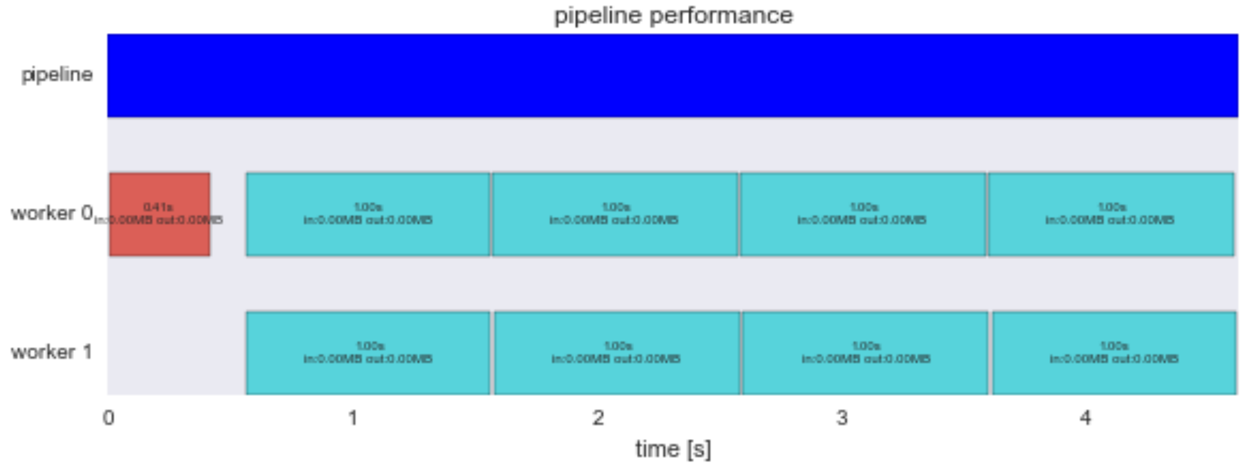


Performance observer

```
from yamal.observers import PerformanceObserver
from time import sleep

def compute_1sec(input):
    sleep(1)  # fancy computation here

pipeline = [splitter(partial(get_chunked_data, chunks=8)),
            compute_1sec]

observer = PerformanceObserver()
run_pipeline(pipeline, observers=[observer], executor=SubprocessYamalExecuto
r(2))
```

```
observer.create_plot()
plt.tight_layout(pad=0.1)
plt.savefig("performance2.png")
plt.clf()
```

```
<matplotlib.figure.Figure at 0x700cf50>
```



Execution backends

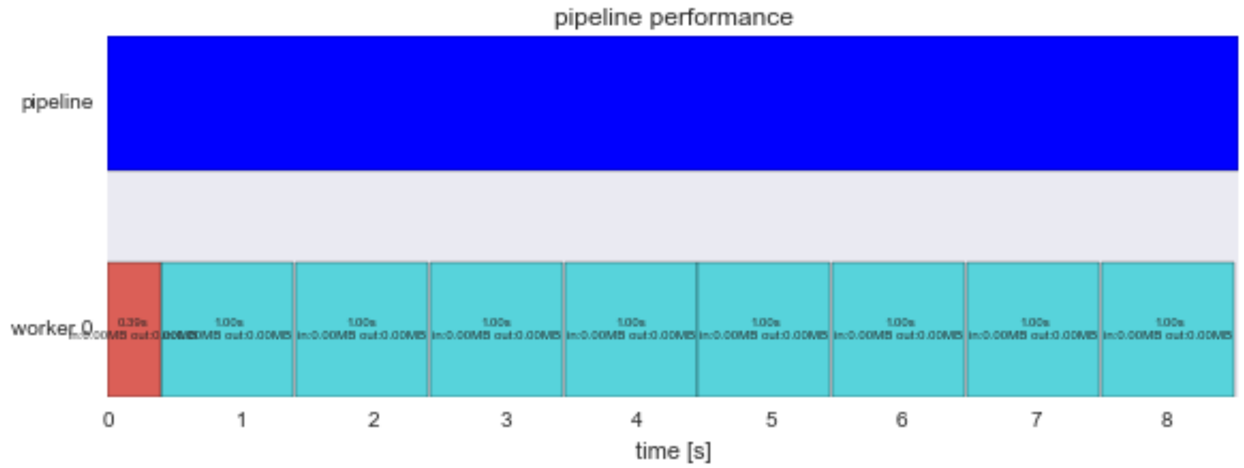
- Local sequential execution
- Local parallel execution in subprocesses
- Remote parallel execution (using internal cluster backend)
- Basically anything that can schedule jobs asynchronously (Spark, distributed, ...)

Sequential execution

```
observer = PerformanceObserver()
run_pipeline(pipeline, observers=[observer])
```

```
observer.create_plot()
plt.tight_layout(pad=0.1)
plt.savefig("performance1.png")
plt.clf()
```

```
<matplotlib.figure.Figure at 0x5ff8990>
```



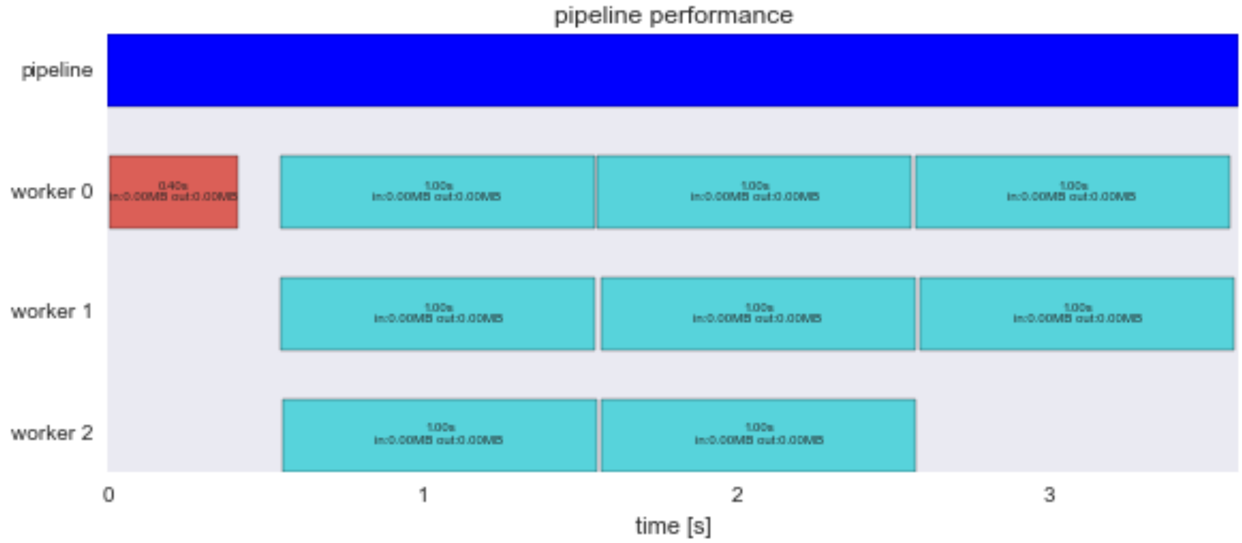
Parallel execution

Single switch for going from sequential to concurrent/distributed:

```
from yamal.subprocess import SubprocessYamalExecutor

observer = PerformanceObserver()
run_pipeline(pipeline, observers=[observer], executor=SubprocessYamalExecuto
r(3))
```

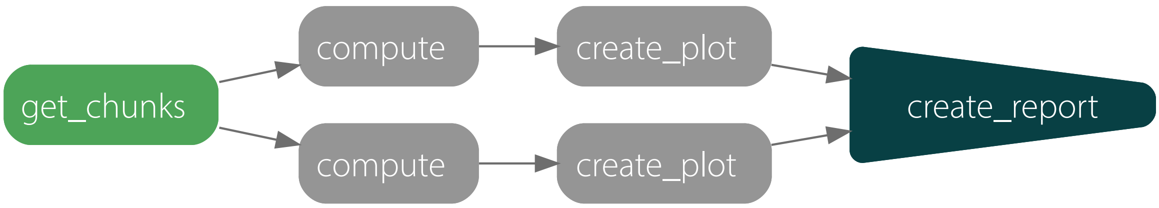
```
observer.create_plot()
plt.tight_layout(pad=0.1)
plt.savefig("performance3.png")
plt.clf()
```

```
<matplotlib.figure.Figure at 0x718f410>
```



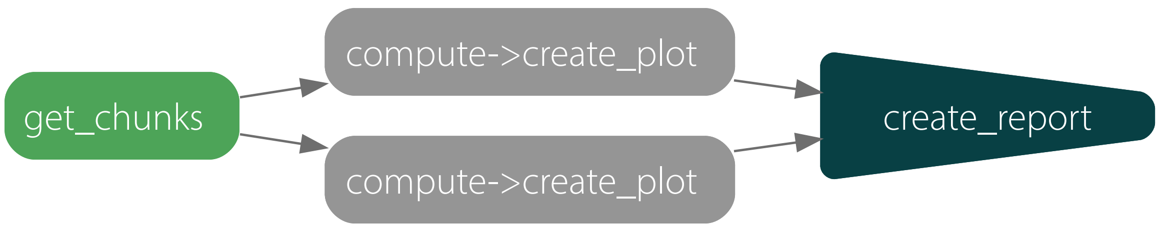
Performance optimizations

```
def get_chunks():
    yield get_chunk(0)
    yield get_chunk(1)

pipeline = [
    splitter(get_chunks),
    compute,
    create_plot,
    reducer(create_report)
]

clear_graph()
run_pipeline(pipeline, suppress_fusing=True)
show_graph()
```



Fusing of linear pipeline segments:

```
clear_graph()
run_pipeline(pipeline)
show_graph()
```

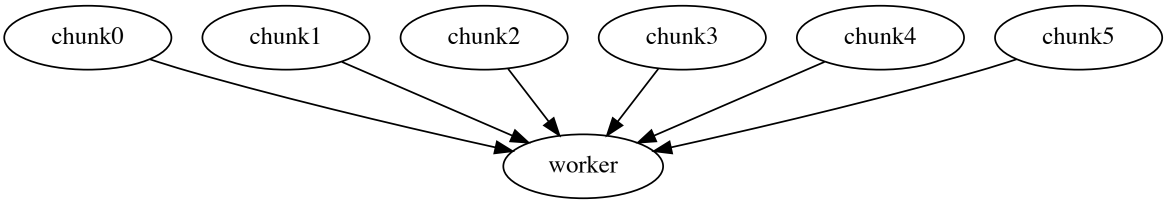


Block size

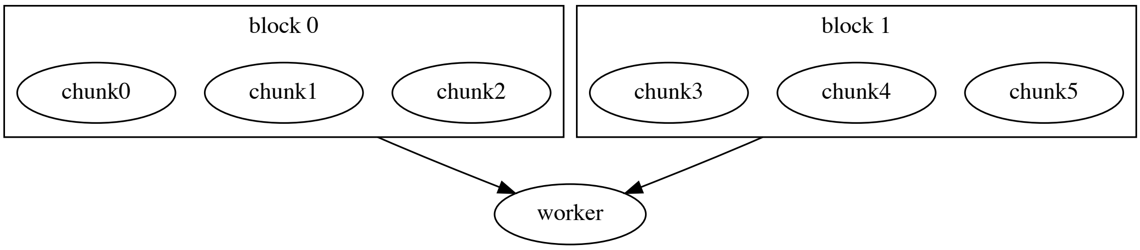
- Scheduling many small jobs can be inefficient
- Allow passing data to workers in larger blocks

```
pipeline = [
    splitter(get_chunks).set(output_blocksize=3)
]
```

```
%dotstr "digraph G {chunk0->worker chunk1->worker chunk2->worker chunk3->wor
ker chunk4->worker chunk5->worker}"
```




```
s = """
digraph G {
  compound=true;
  worker
  subgraph cluster0 {
    label = "block 0";
    chunk2 chunk1 chunk0;
  }
  subgraph cluster1 {
    chunk5 chunk4 chunk3;
    label = "block 1";
  }
  chunk1 -> worker [ltail=cluster0];
  chunk4 -> worker [ltail=cluster1];
}"""
%dotstr s
```



Debugging helpers

- Enter debugger on exception
- Multiprocessing-aware embedding

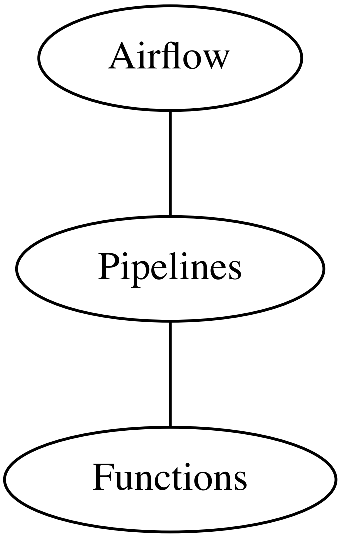
```
def some_function(input):
    from IPython import embed
    embed()
```

Each "embed" acts as a synchronization point.

Part III: Effects on development

Typical project architecture

```
%dotstr "graph G {Airflow -- Pipelines -- Functions}"
```



- Top-level operational view: Airflow
- Intermediate level: Yamal pipelines
- Python functions: Numpy, scikit-learn, matplotlib, ...

Usage of pipelines in projects

- Projects are quite different, monolithic application does not work well
  - Reusable building blocks in core library
  - Each project defines its own I/O and control flow
- Pipelines provide intermediate level of abstraction
  - Project manager can immediately see what is going on
- Not limited to numerical algorithms
  - Can be combined with other tools like Dask

How did this affect our development?

- Incentive for more functional style of programming
    - Developers get concurrency and other nice tools for free
- Overall cleaner architecture
    - Discourages use of spaghetti code and global state
    - More explicit data dependencies
    - Easy to understand
- Improved reusability and testability

Testability

1. Unit tests for individual functions
2. Component tests for pure pipelines without I/O
3. Integration tests for pipelines including I/O

```
pure_pipeline = [compute, create_plot]

pipeline_with_io = [get_data] + pure_pipeline + [write_report]
```

1+2 covers most aspects so that expensive I/O tests can be reduced to a minimum.

Summary

- Separation of pipeline declaration and execution
- More functional style of programming
- Functions do not depend on Yamal
- Control flow based on labels
- Completely data-agnostic (e.g., not limited to numerical problems)
- Other scheduling libraries could act as backends

Outlook

- Explicit annotation of I/O operations
    - Make side-effects explicit
    - Additional documentation for developer
- Tools for debugging in distributed production-like systems
- Optimizations at the pipeline level (e.g., caching)

Thank you! Questions?