University of BRISTOL

Science and Technology Facilities Council

SWIFT HEP

SoftWare InFrastructure and Technology for High Energy Physics

SWIFT-HEP
WP5 Analysis Systems
WP1 Data Management

Sam Eriksen & Timothy Noble

12th November 2024

Overview

- WP5 (Analysis Systems)
  - overview + roadmap
  - Current progress in WP5
  - Future for WP5
- WP1 (Data + workflow management)
  - overview
  - Current progress in WP1
  - Future for WP1

Previous updates

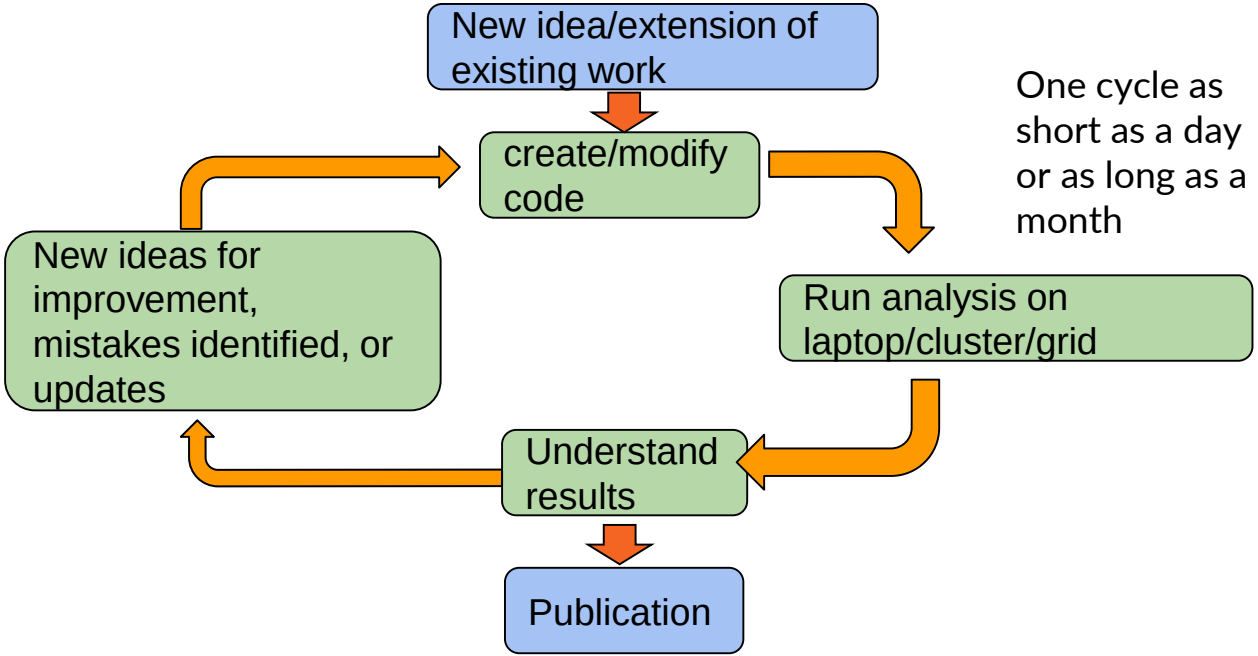
- March 2024
- November 2023
- September 2023
- May 2023
- March 2023
- February 2023

WP5: Analysis Systems

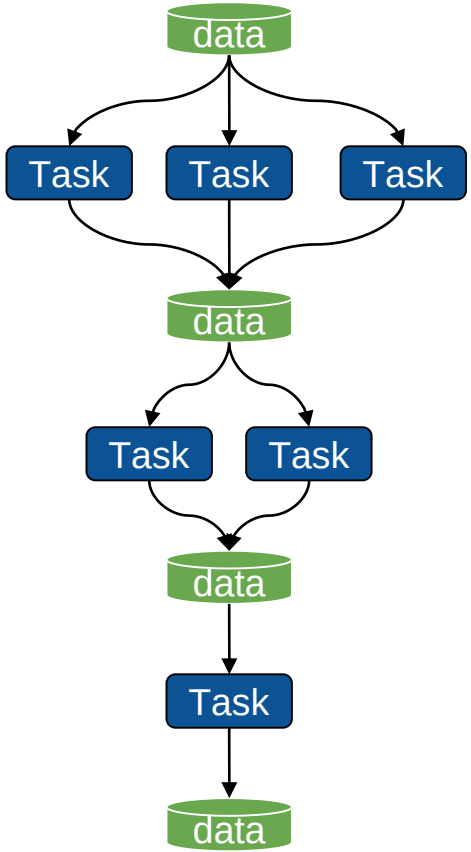
WP5: Analysis Systems

Run analysis workloads optimally on distributed resources

Analysis Anatomy

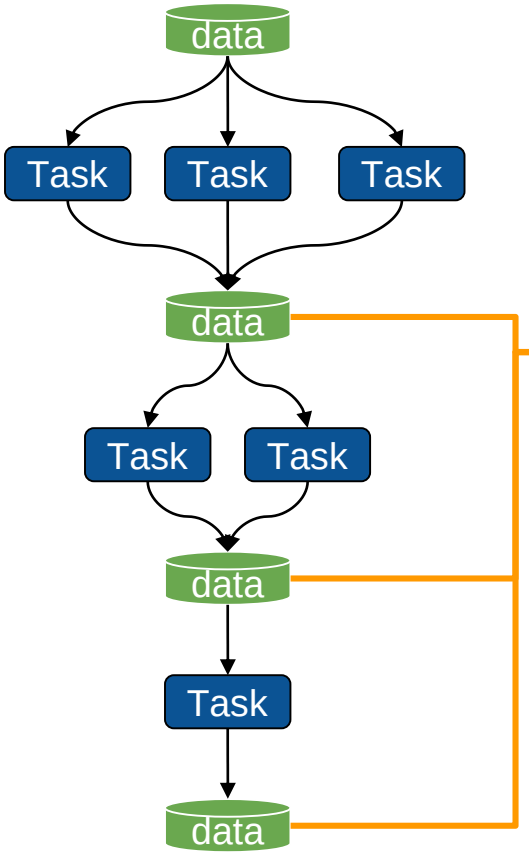


One cycle as short as a day or as long as a month



See talk by Luke Kreczko for some **BIG** Picture

Analysis Anatomy



WP5

WP1

Analysis **step** output

caching

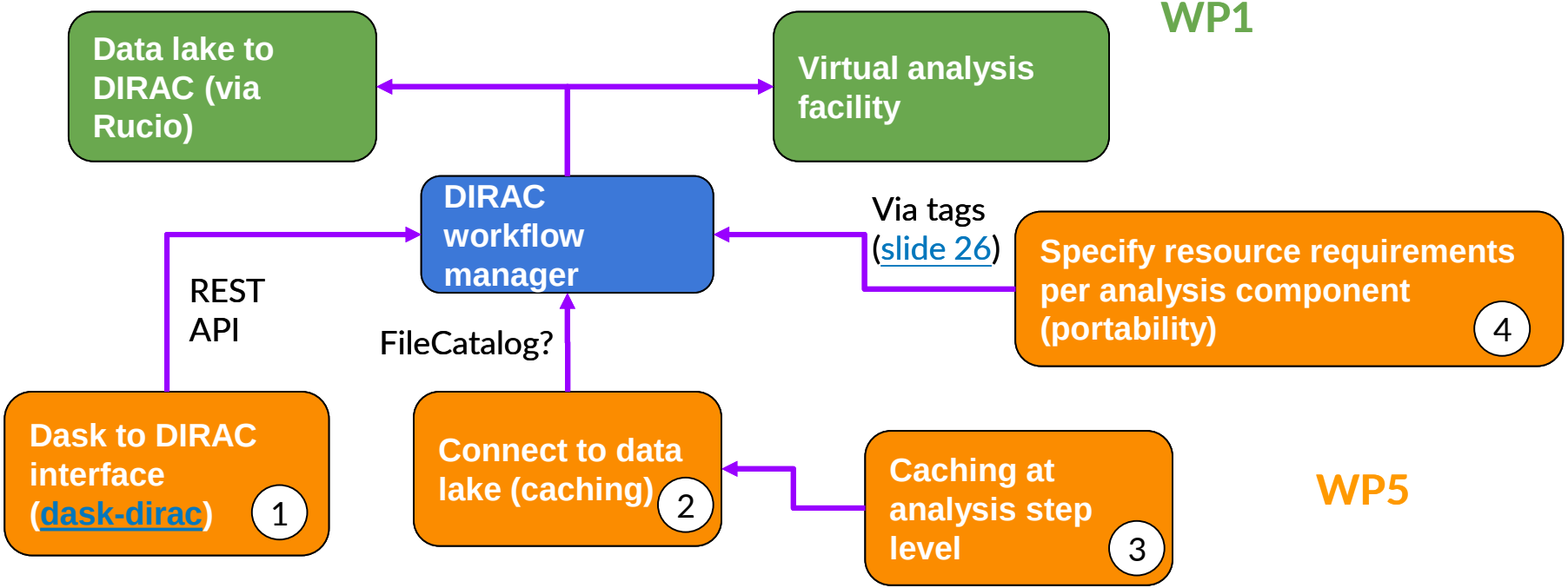
DIRAC

Data lake

WP5 in a nutshell:
Run analysis workloads optimally** on distributed (GridPP) resources

** balanced between user-experience and computing efficiency

WP5: Roadmap



More detailed slides

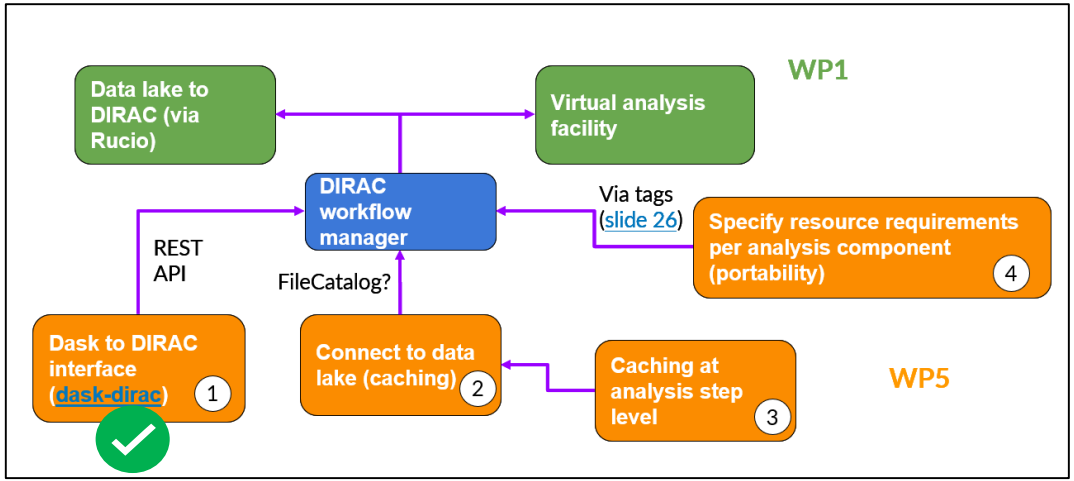
WP5: Roadmap

1. **Dask to DIRAC interface (dask-dirac)**
   - Add extension to dask
   - Dask is able to parallelize any python code
2. **Connect to data lake (caching)**
   - Add the ability to save output after dask instance has closed
3. **Caching at analysis step level**
   - Avoid having to re-run analysis steps
   - Persistent caching
4. **Specify resource requirements per analysis component (portability)**
   - E.g. Let some stages run on GPUs

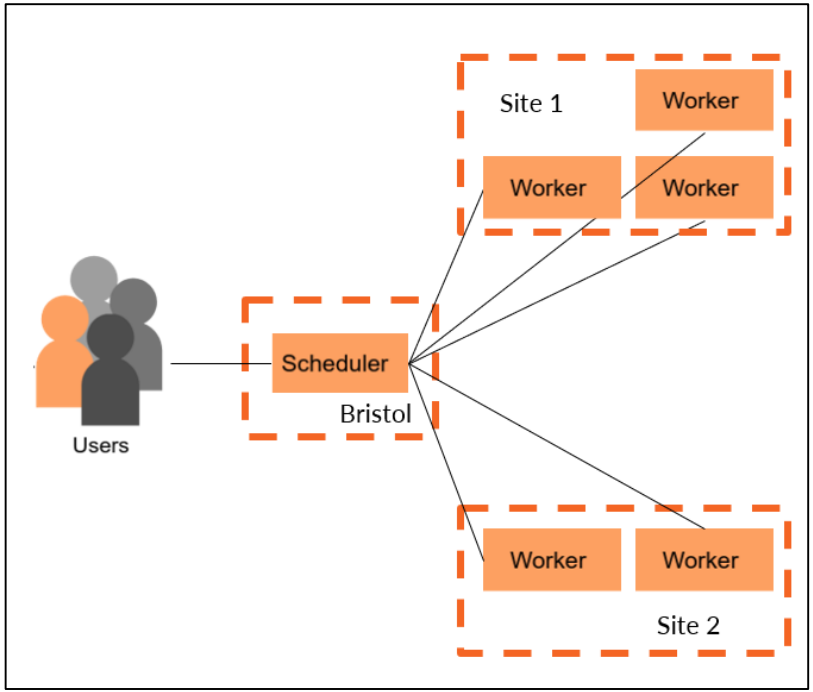
See talk by Luke Kreczko for some future planning

Where did we leave things at the last workshop

Where we left things last time



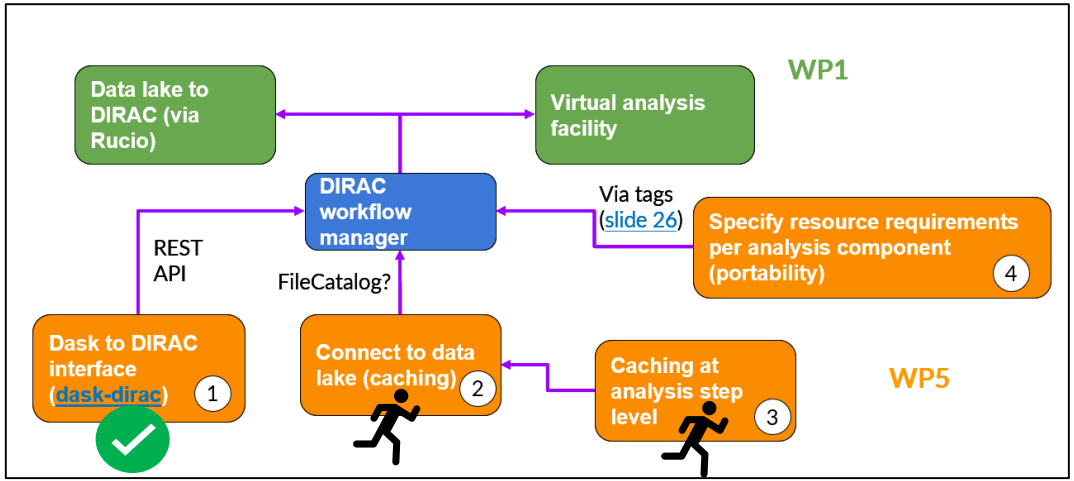
Dask to DIRAC interface (dask-dirac)



- Reliable dask-DIRAC interface
- Multi-site submission / fine control possible on distribution of workers
- Integrated with CMS AGC

```
# Local workers
cluster = LocalCluster()
# PBS
cluster = PBSCluster()
# Dirac workers
cluster = DiracCluster()
```

Where we left things last time



Connect to data lake

- Interact with diracdev StorageElement via HTTP + x509
- gfal underneath
- Painful and wanted to move away from
- RUCIO via tokens was ideal

Caching at analysis step level

- Began looking at intermediate result caching using scheduler plugins
- Allows for tasks to be run when a worker connects/disconnects or transitions (e.g. from running to memory) a job
- Idea: perform check (ideally via RUCIO) if result exists before a task starts

What's happened since the last workshop

What have we been focusing on

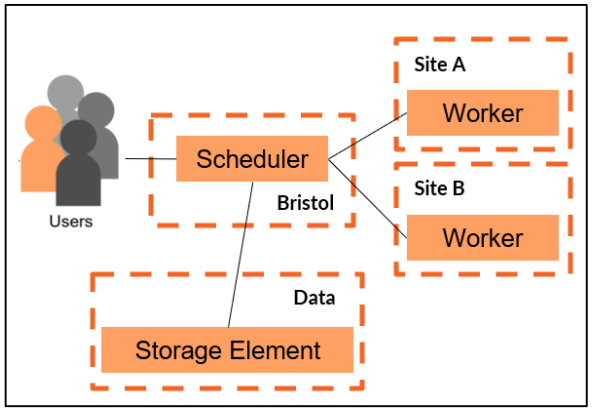
Connect to data lake

- Wanted to communicate with RUCIO via REST-API with tokens
  - Able to interact with RUCIO, but only with x509
  - Tokens generated from the OIDC-Agent and Rucio at least with the IRIS-IAM varied, to the point where Rucio would not accept the OIDC-Agent tokens. Investigation underway in association with James Walder from SKAO (Full token stack Rucio instance)
  - Token Authentication in Rucio is being redesigned by a token auth SIG
- We've parked progress on this until Tim has investigated further

What have we been focusing on

Caching at analysis step level

Initial idea; dask plugins

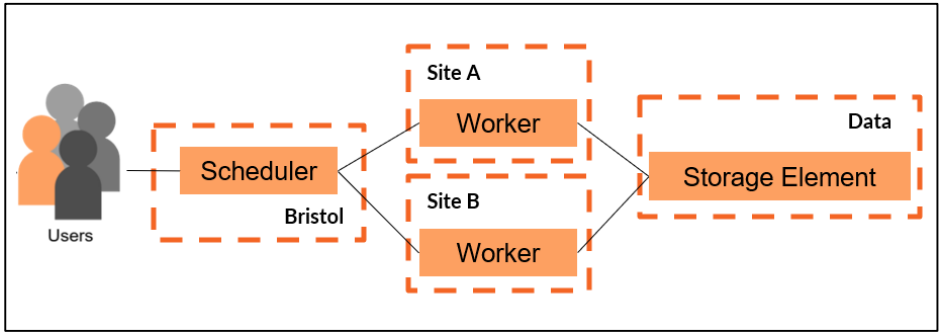


Scheduler Plugins

- Allows you to run custom Python code on scheduler when certain things happen
- Example;
  - Scheduler checks if output exists; if exists, load and don't run job, else, run job as normal
  - Workers send all data back to Scheduler which then perform 'saves'
  - Essentially intercepts result before it's passed to the user

Worker Plugin

- Allows you to run custom Python code on all workers at certain event in the worker's lifecycle (such as when a process finishes)
- Example;
  - Workers check if output exists; if exists, load and don't run job, else run job as normal
  - In this case, workers interact with storage



Note

- Dask already does caching which does similar to the above, but we want **persistency between dask sessions**
- There's also the option for scheduler and worker plugins together

What have we been focusing on

Caching at analysis step level

Initial idea; dask plugins

Worker Plugin – where did we get to

- Created examples to explore what information we can get from the worker at each stage
- Created worker plugin to show what in the worker directory

Problem

- This is only done when the worker is killed (or the plugin removed)
- Limited by access to analysis step information / would require interrupting the processes
- Very difficult to alter task graph on the fly

```python
from distributed.diagnostics.plugin import WorkerPlugin
from distributed.diagnostics.plugin import forward_stream
import contextlib
import os

class ForwardOutput(WorkerPlugin):

    tabnine: test | explain | document | ask
    def setup(self, worker):
        self._exit_stack = contextlib.ExitStack()
        self._exit_stack.enter_context(forward_stream("stdout", worker=worker))
        self._exit_stack.enter_context(forward_stream("stderr", worker=worker))

    tabnine: test | explain | document | ask
    def teardown(self, worker):
        print("{}: {} ".format(os.getcwd(), os.listdir(os.getcwd())))
        self._exit_stack.close()
```

```
plugin = ForwardOutput()
✓ 0.0s

client.register_plugin(plugin, name='my-plugin')
✓ 0.0s

{'tcp://130.246.45.124:50000': {'status': 'OK'}}
```

```
# remove plugin
client.unregister_worker_plugin(name='my-plugin')
✓ 0.0s

/scratch/condor/dir_112165/406MDmIA544n8FVDjqQVj3jqav4PzpABFKDmon9MDmXIFKDmzPLhKm/DIRAC_mMYY6bpilot/1300: ['job.info', 'std.err', 'std.out']
2024-03-14 19:01:02,044 - distributed.worker - INFO - Removing Worker plugin my-plugin

{'tcp://130.246.45.124:50000': {'status': 'OK'}}
```

What have we been focusing on

Caching at analysis step level

Initial idea; [dask plugins]

Scheduler Plugin – where did we get to

- Created examples to explore what information we can get from the scheduler at each stage
- Created example to check if a task (via hash) had already been run (in backup)

```python
class MyPlugin(SchedulerPlugin):
    def __init__(self):
        self.task_counter = 0

    def transition(self, key, start, finish, *args, **kwargs):
        if start == 'processing' and finish == 'memory':
            self.task_counter += 1

    def get_task_count(self):
        return self.task_counter
```

Executed at 2024.03.11 08:55:44 in 3ms

```python
for i in range(10):
    x = client.submit(lambda x: x + 1, i)
    result = x.result()
    print(i, plugin.task_counter)
```

Executed at 2024.03.11 08:58:34 in 121ms

```
0 16
1 17
2 18
3 19
4 20
5 21
6 22
7 23
8 24
9 25
```

Workflow would be;

- Some kind of combination of scheduler and worker plugin
- Access task hash, check if it exists, and then replace with loading or saving data

What have we been focusing on

Caching at analysis step level

Initial idea; dask plugins

Problems with this approach

- Very hard to manipulate task graphs in this approach; hashes associated with tasks are only semi-deterministic -> change between dask sessions for the same code
- Very difficult to access individual steps in a task graph. Very coarse; approach was more limited to final task.

Solution

- Work on calculating hash that is independent of scheduler state and without plugins

deterministic hash test

```
outputs = client.map(neg, [1,2,3])
✓ 0.0s

-----------------------------------------
Current time: 20:02:44
Tasks submitted: ['neg-cb23b95d4d96dbbf89cd1d672843469a'
```

```
outputs = client.map(neg, [1,2,3])
✓

-----------------------------------------
Current time: 20:04:50
Tasks submitted: ['neg-3c404328af689f5d903f5dda07c47bd2'
```

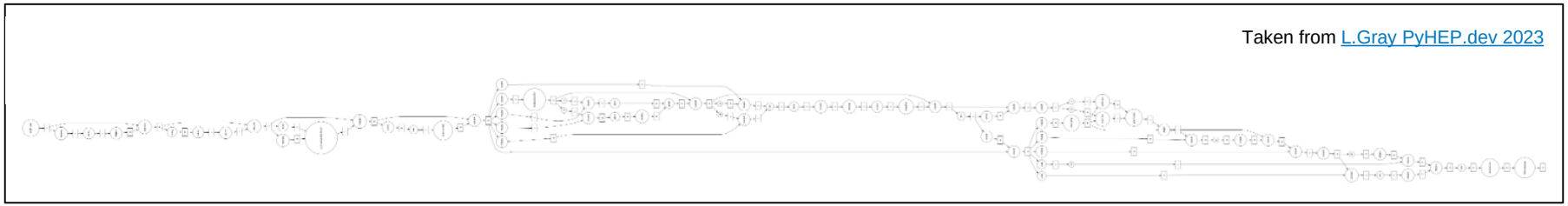
What have we been focusing on

Caching at analysis step level

Ways to interact with dask submission

- client.get
- client.map (this is what CMS AGC uses)
- client.submit
- client.compute

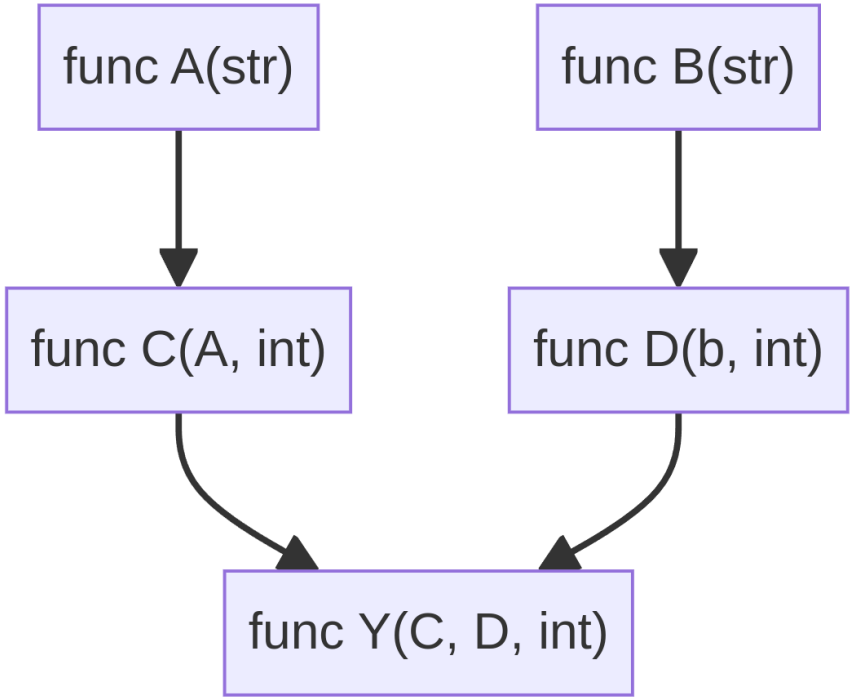
- Find a common area where we can intercept the task graph and manipulate it
- Found _graph_to_futures* where we have access to the full graph



*Actually, started with collections_to_dsk which only works with client.compute and developed a lot of this there before changing to _graph_to_futures

What have we been focusing on

Caching at analysis step level



Scenarios to take into account for a demonstrator

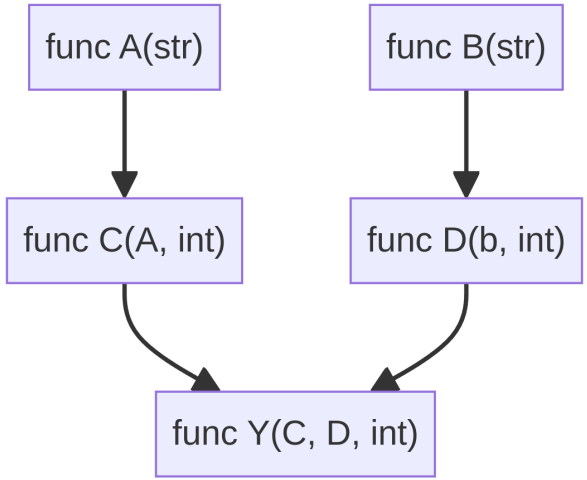
1. Nothing changes -> read & return Cache(Y)
2. A changes -> read Cache(D), calculate the rest
3. D changes -> read Cache(C, B), calculate the rest
4. Y changes -> read Cache(C, D), calculate Y

Requires

1. Probing task graph to work out output / input
2. Checking storage area for if it exists
3. Insert loading / saving stages into graph

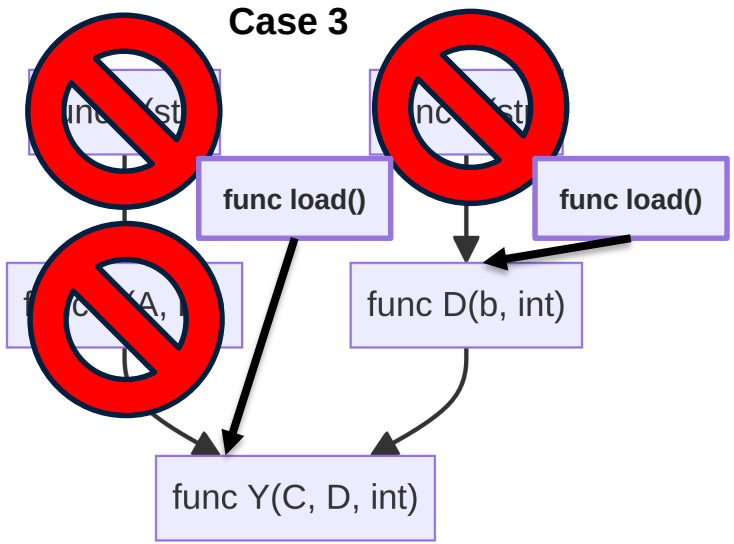
What have we been focusing on

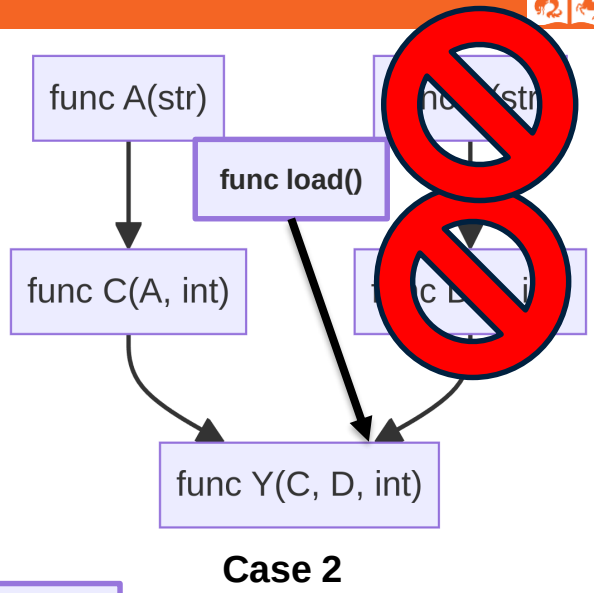
Caching at analysis step level



Scenarios to take into account for a demonstrator

1. Nothing changes -> read & return Cache(Y)
2. A changes -> read Cache(D), calculate the rest
3. D changes -> read Cache(C, B), calculate the rest
4. Y changes -> read Cache(C, D), calculate Y





What have we been focusing on

Caching at analysis step level

Development State

- We've created a new dask client (DiracClient) which adds persistent caching functionality

```
For task in Tasks:
    hash = calculate_task_hash(task)
    hash_found = check_storage(hash)
    if hash_found:
        task = load_from_storage(hash)
    else:
        task = task + write_to_storage(hash)
```

```python
class DiracClient(Client):
    """Client for caching dask computations"""

    def _graph_to_futures(
        self,
        dsk: dict[str, Any] | HighLevelGraph,
        *args: dict[str, Any],
        **kwargs: dict[str, Any],
    ) -> Any:
        if not isinstance(dsk, HighLevelGraph):
            dsk = HighLevelGraph.from_collections(id(dsk), dsk, dependencies={})

        info = dsk.to_dict()
        logging.debug(
            "Input dask graph:\n%s\n---------\nperforming caching checks\n---------",
            info,
        )
```

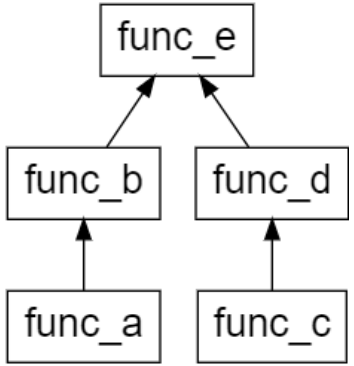
Additional functionality in the works/what needs more thought;

- Changeable location of Persistent Cache; want RUCIO and local options at the very least
- More work is needed to be compatible with CMS ACG; should wait for coffea-2024
- Default is to cache everything, good option to have but likely no ideal

What have we been focusing on

Caching at analysis step level

```python
def func_a(x):
    return x**2

def func_b(x):
    return x**3

def func_c(x):
    return x + 120

def func_d(x):
    return x + 14

def func_e(x, y):
    return x - y
```



Example of what the output is from simple 5-function graph

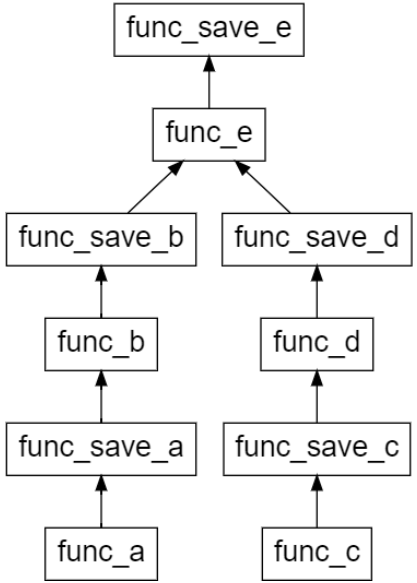
Path 1
first time running

Path 2
second run, but function d has been edited

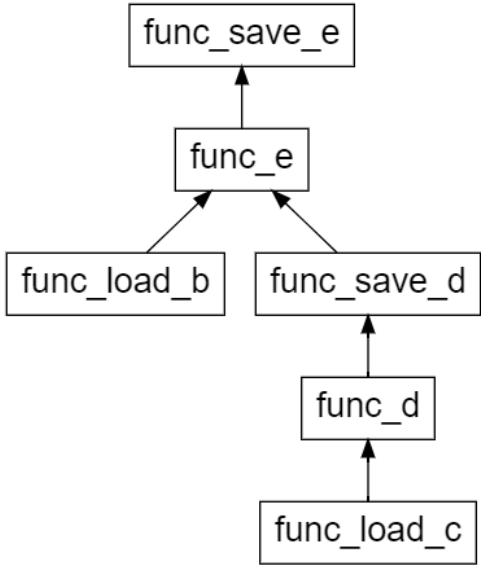
Path 1

Path 2

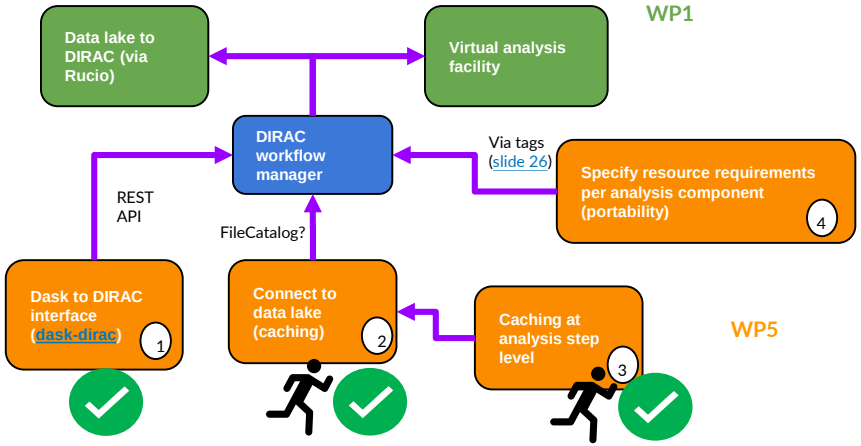
Case if no cache entries have been found



Case if cache b and cache c are found



Summary and plan going forward



- We have a dask-dirac interface
  - Workers are split across sites
  - Effectively splits up work across these sites
- We have been looking at task graph manipulation
  - Found entry point giving fine-grain access to a graph
  - Implemented caching that persists between dask sessions
  - Avoid rerunning tasks where result exists
- Where are we going
  - Awaiting tokens
  - Awaiting ACG update -> coffea.2024

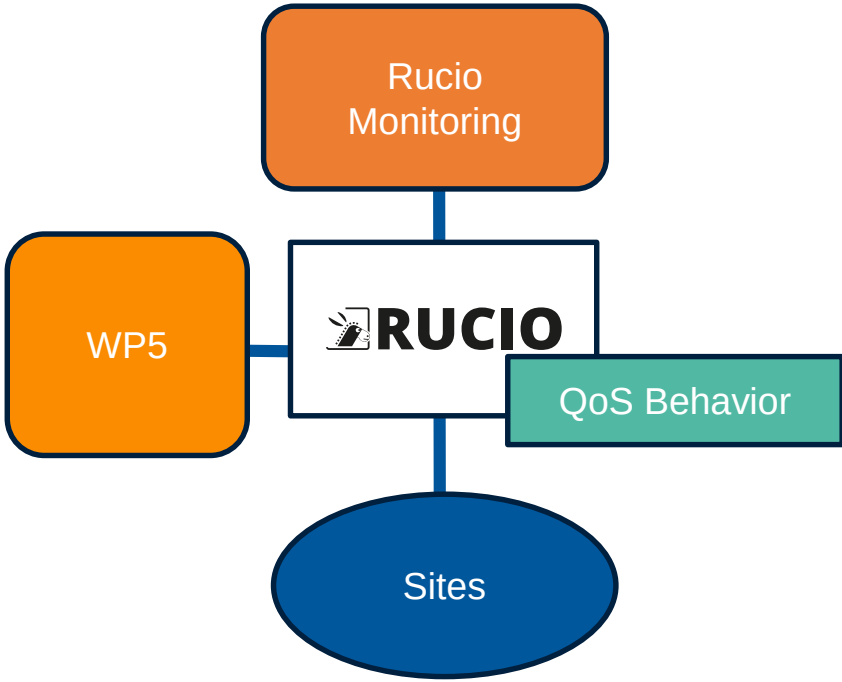
WP1: Data Management

WP1: Data Management

Optimise the heterogeneous storage infrastructure across the UK

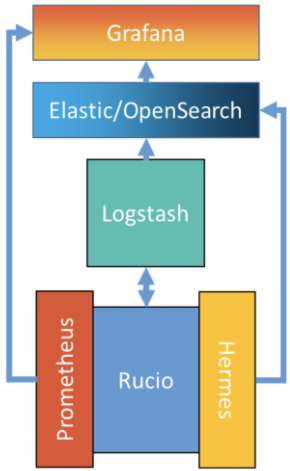
WP1: Data Management Overview

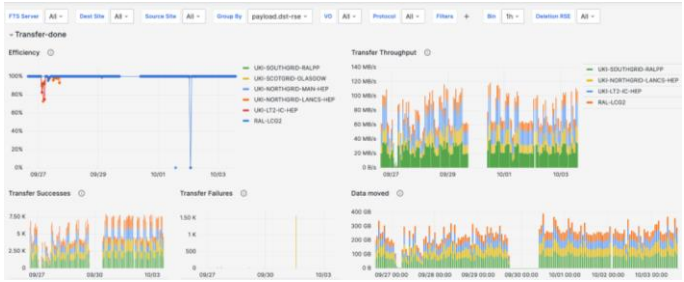
- Deploy a UK based prototype data lake
  - Core sites
  - S3 sites
  - State-less storage
- Generate metrics for comparison of current to improvements made
- Implement Quality of Service behaviour and information in Rucio
  - Develop Rucio to expand the levels of service it provides
  - Create new behaviours in Rucio to enable improved data access
- Produce site recommendations on how to optimise data access and stateless storage
- Setup and test SSD storage endpoints to test and develop fast storage endpoints

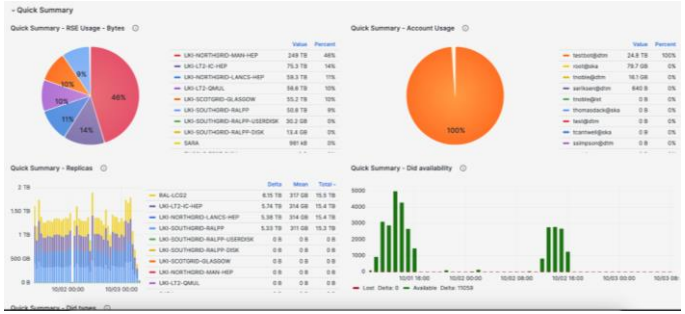


Data Management developments

- Tim has redesigned and expanded upon previously shown monitoring solution to be production-ready (Deployed for astronomy experiment LSST), deployed with OpenSearch.
    - made these tools available to other communities by contributing deployment back to Rucio repositories (Rucio workshop talk)
- Takes 3 different data streams from Rucio, to be visualised in Grafana.
- Minimal design to ensure easy to maintain and deploy
- Allows for quantification of future developments

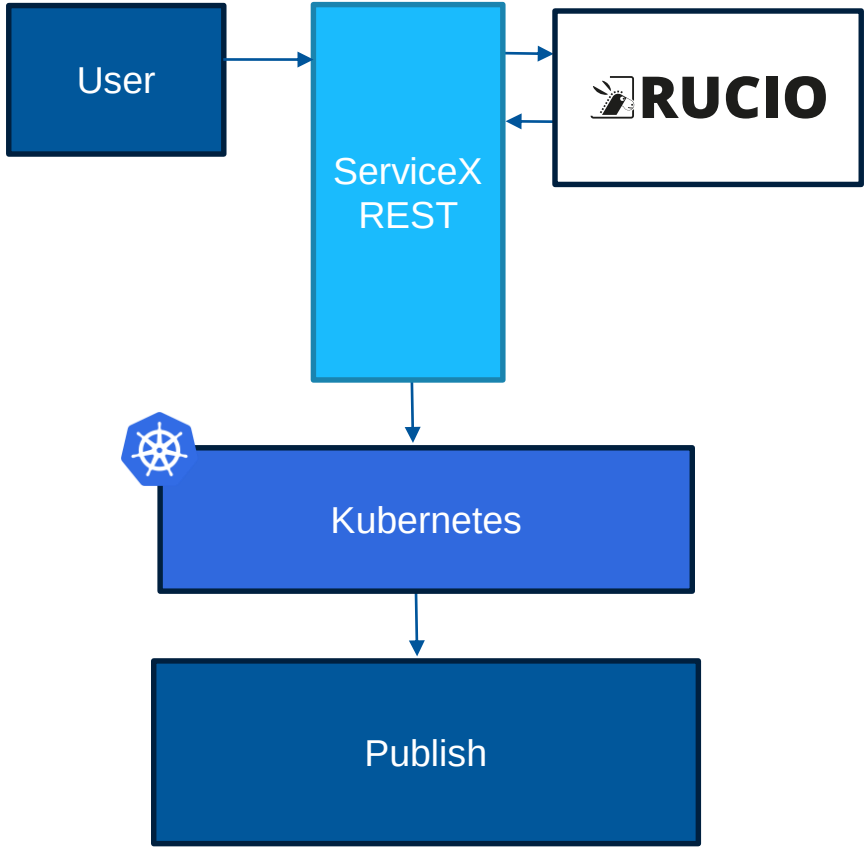






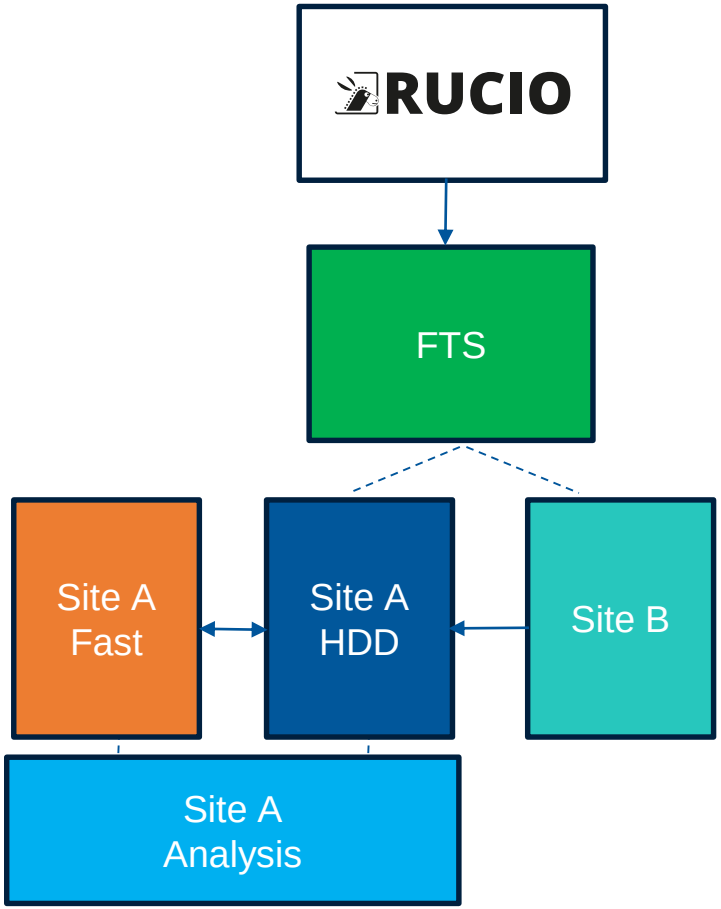
Data Management development plans

- Investigation of ServiceX (or other tools to optimise data access for analysis)
  - Developed by IRIS-HEP
  - Software that sits between the Storage and Analysis Facility (close to storage), and extracts a subset of data columns from whole files – rather than moving the whole file
  - Reduces the networking needs from storage to analysis
    - User requests a subset of data from file(s)
    - ServiceX queries Rucio for files
    - Job started in Kubernetes cluster to extract needed files and create a new one

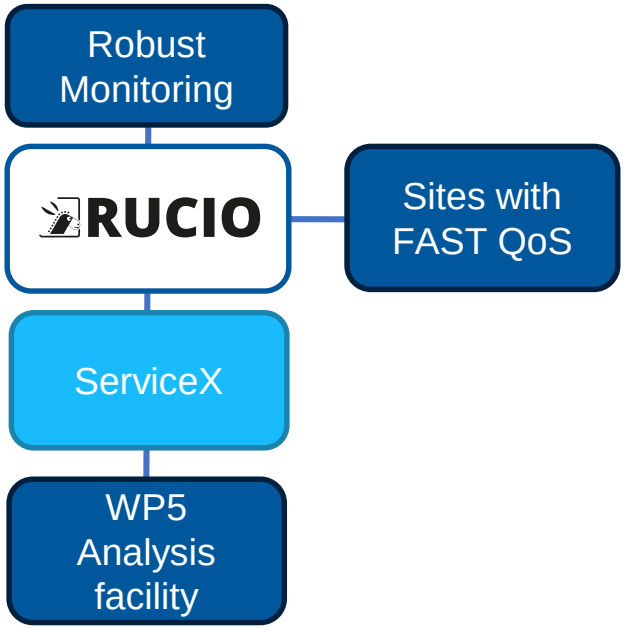


Data Management development plans

- Fast Storage
  - Sites provision Storage by Capacity, but not throughput or access rates
    - Issues will arise with larger data files and increased required data rates where storage endpoints provision more storage, but throughput does not scale at the same rate
  - Further coordination with Core Rucio team to develop Quality of Service to better serve Fast storage
  - Using file popularity to move data between Fast and Disk storage endpoints to ensure read/write needed for analysis



Data management summary and future plan

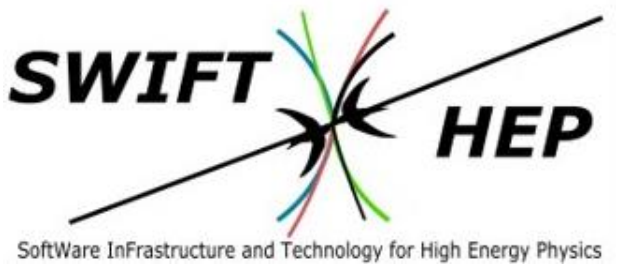


- Deployed Robust monitoring at RAL for SWIFT-HEP and at Stanford for LSST (improvements to one feed into the other)
- Approaching data management from Software and Hardware utilisation
- Further integrations with the Analysis facility to enable better data access
- Will develop Rucio to investigate the prioritization of fast storage as more fast storage pools are available in the UK now
- Will work with CERN liaisons to test deployment ServiceX at RAL to test data access improvements


University of
BRISTOL


UKRI
Science and
Technology
Facilities Council

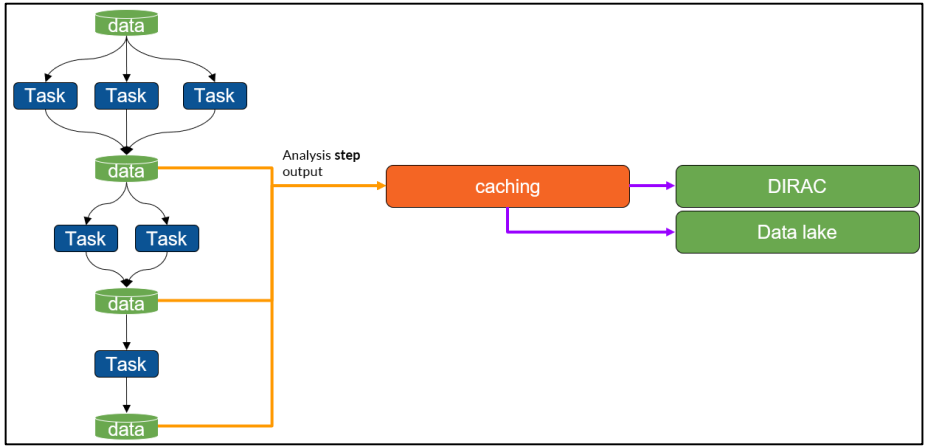
Questions?


SWIFT
HEP
SoftWare InFrastructure and Technology for High Energy Physics

BACKUP

What have we been focusing on

Caching at analysis step level



What do we want to demonstrate

- Imagine a complex task that you want to run.
- If you know you are going to have to run this multiple times, but are only modifying one of the functions, you can think of a few ways to reduce the reprocessing;
- Most obvious would be to break up the graph and submit each task separately and save the outputs, and if you are rerunning, load the bits you haven't changed.
- We want that to automatically happen when you submit something.

Modifying a task graph: Practical example

Caching at analysis step level

This is what we have to play with;

Task is;

- Calculate hash
- Check if hash exists somewhere
- If exists; replace with load
- If doesn't; add save

hash('func_b', hash('func_a', 2))

```
{('func_a-func_b', 0): (<function func_b at 0x7f5d5a957760>,
                        (<function func_a at 0x7f5d5a9576d0>, 2)),
 ('func_b', 0): ('func_a-func_b', 0),
 ('func_c-func_d', 0): (<function func_d at 0x7f5d5a957880>,
                        (<function func_c at 0x7f5d5a9577f0>, 2)),
 ('func_d', 0): ('func_c-func_d', 0),
 ('func_e', 0): (<function func_e at 0x7f5d5a957910>,
                 ('func_d', 0),
                 ('func_b', 0))}
```

Same as above

hash('func_d', hash('func_c', 2))

Same as above

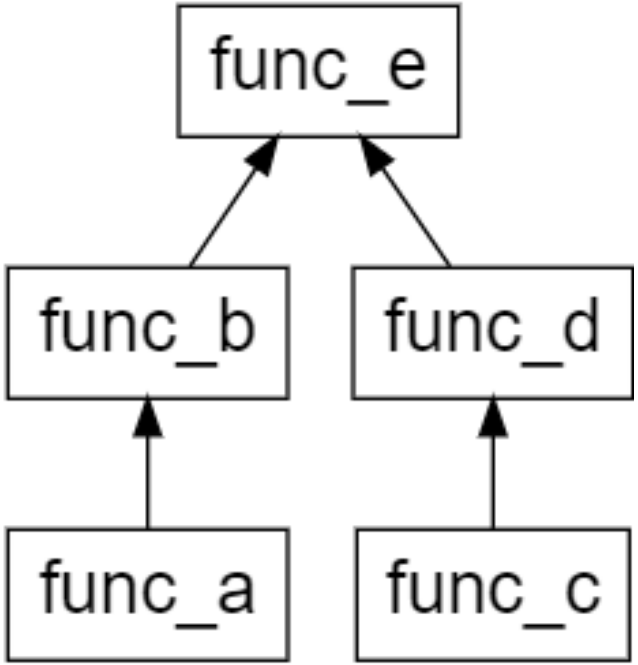
hash('func_e', hash('func_d', hash('hash_c', 2)), hash('func_b', hash('func_a', 2)))

Modifying a task graph: Practical Example

Caching at analysis step level

```python
def func_a(x):
    return x**2

def func_b(x):
    return x**3

def func_c(x):
    return x + 120

def func_d(x):
    return x + 14

def func_e(x, y):
    return x - y
```

```python
# define highlevelgraph
param_a = pd.DataFrame(np.array([2]))
param_b = pd.DataFrame(np.array([2]))

layers = {
    "func_a": {
        ("func_a", 0): (func_a, param_a),
    },
    "func_b": {
        ("func_b", 0): (func_b, ("func_a", 0)),
    },
    "func_c": {
        ("func_c", 0): (func_c, param_b),
    },
    "func_d": {
        ("func_d", 0): (func_d, ("func_c", 0)),
    },
    "func_e": {
        ("func_e", 0): (func_e, ("func_d", 0), ("func_b", 0)),
    },
}
dependencies = {
    "func_a": set(),
    "func_b": {"func_a"},
    "func_c": set(),
    "func_d": {"func_c"},
    "func_e": {"func_b", "func_d"},
}

hlg = HighLevelGraph(layers, dependencies)
```



Practical Example

Caching at analysis step level

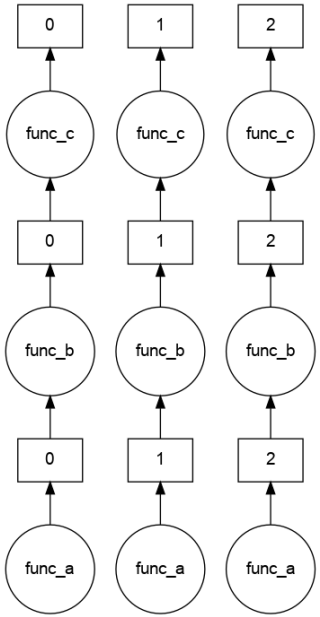
```
{('func_a-func_b', 0): (<function func_b at 0x7f99bb6d3880>,
                        (<function func_a at 0x7f99bb6d37f0>, array([2]))),
 ('func_b', 0): ('func_a-func_b', 0),
 ('func_c-func_d', 0): (<function func_d at 0x7f99bb6d3a30>,
                        (<function func_c at 0x7f99bb6d39a0>, array([2]))),
 ('func_d', 0): ('func_c-func_d', 0),
 ('func_e', 0): (<function func_e at 0x7f99bb6d3ac0>,
                 ('func_d', 0),
                 ('func_b', 0))}
```

```
{('func_a-func_b', 0): (<function func_save at 0x7f99bb6d3be0>,
                        'f84d1e9e761b9b7229bafdf522f347f04b88a65b40568d76f5402ad094b768fd101f3ae957876f0e7bece55676ca102e',
                        (<function func_b at 0x7f99bb6d3880>,
                         (<function func_save at 0x7f99bb6d3be0>,
                          '0776ff206989787bf5a95a35b33716b797efcded6bf71a9e75c1897c5a7ea9ca4f9fb654fe3eba1ebcec7d7911b2dd16',
                          (<function func_a at 0x7f99bb6d37f0>, array([2])))))),
 ('func_b', 0): ('func_a-func_b', 0),
 ('func_c-func_d', 0): (<function func_save at 0x7f99bb6d3be0>,
                        '4c082d6b9b9b918689b81a761abbaae92041057140553ca83e03c4665445a3e78b460e9f0eb8b205e0ef8cb4dd5567f7',
                        (<function func_d at 0x7f99bb6d3a30>,
                         (<function func_save at 0x7f99bb6d3be0>,
                          'f12ce0d8cd6105959827f329b79f2baca8767a4b436cdd843cf008ca50b0ae7754317a320740ecc96d739668308c9d2e',
                          (<function func_c at 0x7f99bb6d39a0>, array([2])))))),
 ('func_d', 0): ('func_c-func_d', 0),
 ('func_e', 0): (<function func_save at 0x7f99bb6d3be0>,
                 '4baeda9eda3c05b98d14b603cea2833cc5b4a3bf1e30cfca90dcaf4f24c9079ea33349840fb93b07c3585cf685dfb2d9',
                 (<function func_e at 0x7f99bb6d3ac0>,
                  ('func_d', 0),
                  ('func_b', 0)))}
```

Practical Example

Caching at analysis step level

```
# Create a Dask Array from the HighLevelGraph
array = Array(hlg, "func_c", shape=(3,), dtype=list, chunks=(1,))
```




```
{('func_a-func_b-func_c', 0): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 2))),
 ('func_a-func_b-func_c', 1): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 3))),
 ('func_a-func_b-func_c', 2): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 4))),
 ('func_c', 0): ('func_a-func_b-func_c', 0),
 ('func_c', 1): ('func_a-func_b-func_c', 1),
 ('func_c', 2): ('func_a-func_b-func_c', 2)}
```

Practical Example

Caching at analysis step level

```
{('func_a-func_b-func_c', 0): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 2))),
 ('func_a-func_b-func_c', 1): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 3))),
 ('func_a-func_b-func_c', 2): (<function func_c at 0x7efdccb7f6d0>,
                               (<function func_b at 0x7efdccb7f640>,
                                (<function func_a at 0x7efdccb7f5b0>, 4))),
 ('func_c', 0): ('func_a-func_b-func_c', 0),
 ('func_c', 1): ('func_a-func_b-func_c', 1),
 ('func_c', 2): ('func_a-func_b-func_c', 2)}
```

```
{('func_a-func_b-func_c', 0): (<function func_c at 0x7fd85844cdc0>,
                               (<function func_b at 0x7fd85844cd30>,
                                (<function func_a at 0x7fd858075870>, 2))),
 ('func_a-func_b-func_c', 1): (<function func_c at 0x7fd85844cdc0>,
                               (<function func_b at 0x7fd85844cd30>,
                                (<function func_a2 at 0x7fd85844c9d0>,))),
 ('func_a-func_b-func_c', 2): (<function func_c at 0x7fd85844cdc0>,
                               (<function func_b at 0x7fd85844cd30>,
                                (<function func_a at 0x7fd858075870>, 4))),
 ('func_c', 0): ('func_a-func_b-func_c', 0),
 ('func_c', 1): ('func_a-func_b-func_c', 1),
 ('func_c', 2): ('func_a-func_b-func_c', 2)}
```

Worker Plugin example

Caching at analysis step level

Description

- When worker is shutdown/killed or the plugin is disconnected, the directory path and contents are displayed

```python
from distributed.diagnostics.plugin import WorkerPlugin
from distributed.diagnostics.plugin import forward_stream
import contextlib
import os

class ForwardOutput(WorkerPlugin):

    tabnine: test | explain | document | ask
    def setup(self, worker):
        self._exit_stack = contextlib.ExitStack()
        self._exit_stack.enter_context(forward_stream("stdout", worker=worker))
        self._exit_stack.enter_context(forward_stream("stderr", worker=worker))

    tabnine: test | explain | document | ask
    def teardown(self, worker):
        print("{}: {} ".format(os.getcwd(), os.listdir(os.getcwd())))
        self._exit_stack.close()
```

```
plugin = ForwardOutput()
✓ 0.0s

client.register_plugin(plugin, name='my-plugin')
✓ 0.0s

{'tcp://130.246.45.124:50000': {'status': 'OK'}}
```

```
# remove plugin
client.unregister_worker_plugin(name='my-plugin')
✓ 0.0s

/scratch/condor/dir_112165/406MDmIA544n8FVDjqQVj3jqav4PzpABFKDmon9MDmXIFKDmzPLhKm/DIRAC_mMYY6bpilot/1300: ['job.info', 'std.err', 'std.out']
2024-03-14 19:01:02,044 - distributed.worker - INFO - Removing Worker plugin my-plugin

{'tcp://130.246.45.124:50000': {'status': 'OK'}}
```

Worker Plugin example

Caching at analysis step level

Description

- When worker is transitioning between states, the state is printed and the number of transitions are counted

```python
class MyPlugin(WorkerPlugin):
    def __init__(self, logger):
        self.worker = None
        self.logger = logger
        self.n_transitions = 0

    def setup(self, worker):
        self.worker = worker

    def transition(self, key, start, finish, *args, **kwargs):
        self.n_transitions += 1
        print('state: {}    n_transitions: {}'.format(finish, self.n_transitions))
```

```
state: executing    n_transitions: 223
state: memory    n_transitions: 224
state: released    n_transitions: 225
state: forgotten    n_transitions: 226
state: waiting    n_transitions: 227
state: ready    n_transitions: 228
state: executing    n_transitions: 229
state: released    n_transitions: 230
state: forgotten    n_transitions: 231
state: memory    n_transitions: 232
state: waiting    n_transitions: 233
```

Scheduler Plugin example

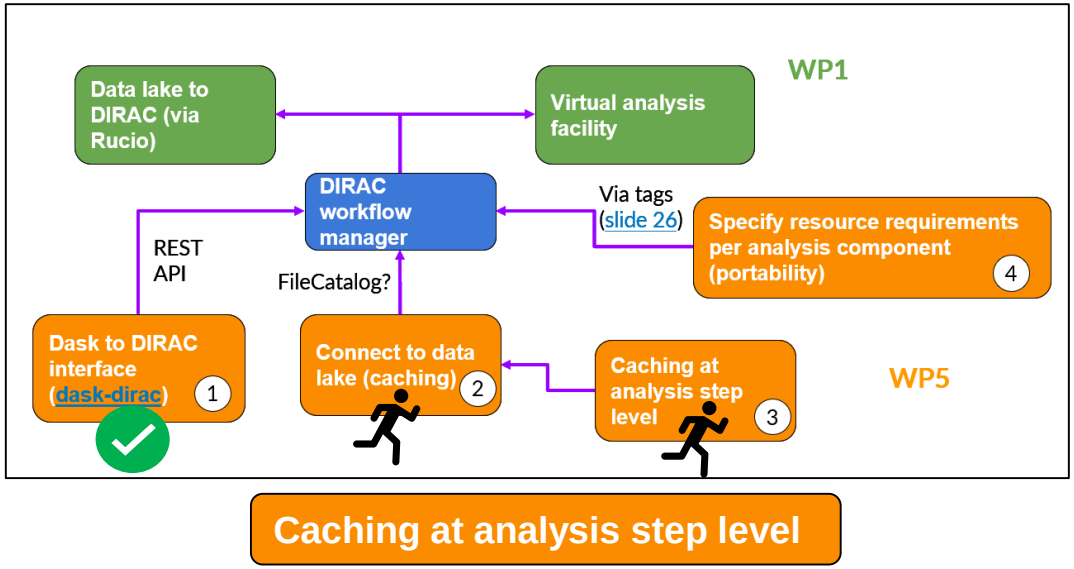
Caching at analysis step level

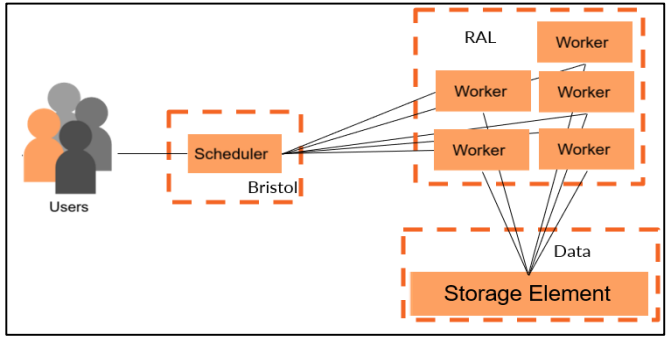
Description

- Remove a task from the graph if it has already been executed

```python
class MySchedulerPlugin(SchedulerPlugin):

    global previous_executed_tasks

    tabnine: test | explain | document | ask
    def update_graph(self, scheduler, keys=None, tasks=None, restrictions=None, **kwargs):
        current_time = datetime.now().strftime('%H:%M:%S')
        print("-------------------------------------------")
        print(f"Current time: {current_time}")
        print(f"Tasks submitted: {tasks}")
        print(f"Prevously Executed Tasks: {previous_executed_tasks}")
        print(f"Keys: {keys}")
        print(f"scheduler tasks: {scheduler.tasks}")
        print(f"scheduler unrunnable: {scheduler.unrunnable}")

        tasks_to_execute = []
        for task in tasks:
            if task in previous_executed_tasks:
                print(f"Task {task} has already been executed - removing from task list")
            else:
                tasks_to_execute.append(task)
                previous_executed_tasks.append(task)

        tasks = tasks_to_execute
        keys = set(tasks)

        print(f"Tasks to be executed: {tasks_to_execute}")
        print(f'keys: {keys}')

        # Update the scheduler's tasks and keys
        scheduler.tasks = {key: scheduler.tasks[key] for key in keys}
        scheduler.unrunnable = {key: scheduler.unrunnable[key] for key in keys if key in scheduler.unrunnable}
        print(f"scheduler tasks: {scheduler.tasks}")
        print(f"scheduler unrunnable: {scheduler.unrunnable}")

    tabnine: test | explain | document | ask
    def transition(self, key, start, finish, *args, **kwargs):
        print("-------------------------------------------")
        print(f"Task {key} is transitioning from {start} to {finish}")
```

Where we left things last time






```python
class MyPlugin(WorkerPlugin):
    def __init__(self, logger):
        self.worker = None

    def setup(self, worker):
        self.worker = worker

    def transition(self, key, start, finish, *args, **kwargs):
        if finish == "memory":
            # Add data
            requests.request("POST", url, json=payload, headers=headers)
```

- Began looking at intermediate result caching using scheduler plugins
- Allows for tasks to be run when a worker connects/disconnects or transitions (e.g. from running to memory) a job
- Idea: perform check (ideally via RUCIO) if result exists before a task starts