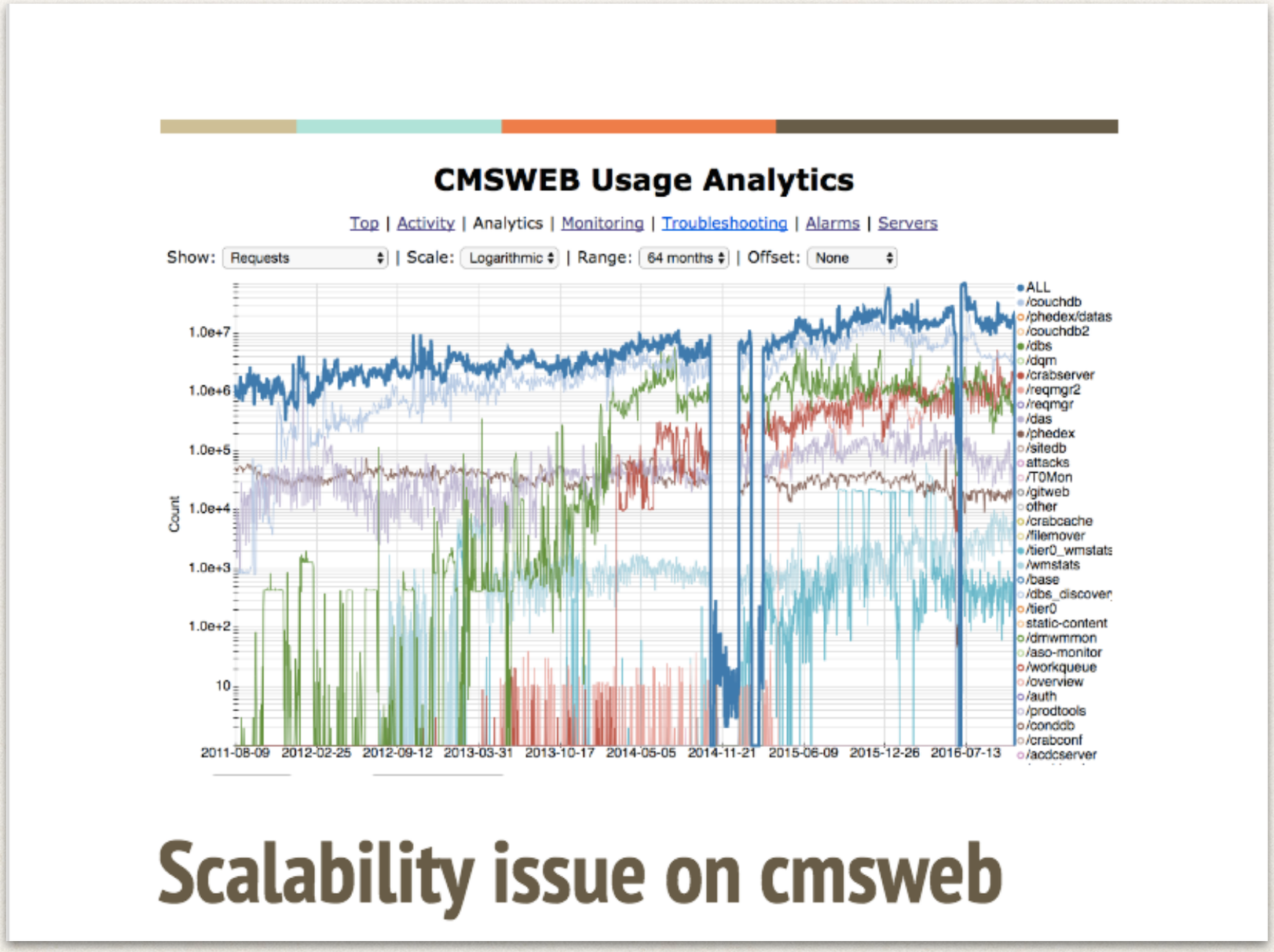
# Concurrency, ML and transfer R&D

Valentin Kuznetsov, Cornell University

*CMS R&D meeting*

# Concurrency R&D




C&O week 2016: http://bit.ly/2n5E36h

# DAS Go client

- DAS Go server & client code is available
- DAS Go client benefits:
  - bypass DAS cache server => speed-up by 10+ times
    - stay on par with DBS APIs, e.g. few seconds to fetch 125K+ files
  - leverage built-in concurrency to make data-service calls (no 3d party lib dependencies)
  - static executable => zero deployment headache
  - runs on OSX/Linux (x86_64/Arm64/Power8 and SL6/SL7)

```
lxplus091(18:48:21) ~ > time ~/public/dasgoclient/dasgoclient_linux -query="file dataset=/ZMM/Summer11-DESIGN42_V11_428_SLH
C1-v1/GEN-SIM" -json
[
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/02ACAA1A-9F32-E111-BB31-0002C90B743A.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/24DAE39C-AC32-E111-AED8-0002C90B743A.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/7E9C0287-0933-E111-8FE5-0002C90B7F90.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/980E0788-0933-E111-9C73-0002C90A3678.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/9CF68F8F-E732-E111-AD97-0002C90B743A.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/A4ADE5A5-0733-E111-ACC2-0002C90A3426.root" ,
"/store/mc/Summer11/ZMM/GEN-SIM/DESIGN42_V11_428_SLHC1-v1/0003/AAF5DDA5-0733-E111-A8D4-0002C90A3426.root"
]

real    0m0.133s
user    0m0.028s
sys     0m0.022s
lxplus091(18:48:31) ~ > time ~/public/dasgoclient/dasgoclient_linux -query="file,run,lumi block=/SingleMu/Run2011B-WMu-19No
v2011-v1/RAW-RECO#19110c74-1b66-11e1-a98b-003048f02c8a"
/store/data/Run2011B/SingleMu/RAW-RECO/WMu-19Nov2011-v1/0002/0031EE0D-661B-E111-B832-001E0B47AC98.root175863[50,54,74,56,73
,55,49,60]
/store/data/Run2011B/SingleMu/RAW-RECO/WMu-19Nov2011-v1/0002/0031EE0D-661B-E111-B832-001E0B47AC98.root175875[24,35,23,36]
```

• • •

```
/store/data/Run2011B/SingleMu/RAW-RECO/WMu-19Nov2011-v1/0002/FEF3962B-6B1B-E111-9F6B-0017A477101C.root175887[90,64,63,87,89
,88,65,66]
/store/data/Run2011B/SingleMu/RAW-RECO/WMu-19Nov2011-v1/0002/FEF3962B-6B1B-E111-9F6B-0017A477101C.root176933[164,169,171,16
2,163,165,168,170]

real    0m2.363s
user    0m0.067s
sys     0m0.083s
```

Available on CVMFS: **/cvmfs/cms.cern.ch/common/dasgoclient**

On 20170118: 7739 das_client, 4819 web and 80658 dasgoclient requests

…

On 20170306: 8405 das_client, 4507 web and 32919 dasgoclient requests

…

On 20170317: 3377 das_client, 3520 web and 159995 dasgoclient requests

# Python async I/O

- Python asyncio provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives; available in python 3.5+

- asyncio server requires an event loop processing various coroutines
  - coroutines and tasks based on yield from (PEP 380), to help write concurrent code in a sequential fashion

- asyncio provides synchronization primitives for use between coroutines in a single thread, mimicking those in the threading module

# Python async I/O client

```
#!/usr/bin/env python3
import asyncio
from aiohttp import ClientSession

async def fetch(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            return await response.read()

async def bound_fetch(sem, url):
    for idx in range(5): # make 5 attempts
        try:
            async with sem:
                return await fetch(url)
        except:
            await asyncio.sleep(0.1)

async def run(loop, nclients, nsem):
    url = "http://localhost:8000/?query=test"
    tasks = []
    sem = asyncio.Semaphore(nsem) # Limit # concurrent requests to process
    for i in range(nclients): # number of concurrent clients
        task = asyncio.ensure_future(bound_fetch(sem, url.format(i)))
        tasks.append(task)
    responses = asyncio.gather(*tasks)
    await responses
    print("Responses:", responses)

def main():
    loop = asyncio.get_event_loop()
    # send 10 clients and constrain to process 1k requests
    future = asyncio.ensure_future(run(loop, 10, 1000))
    loop.run_until_complete(future)

if __name__ == '__main__':
    main()
```

**async** should precede all asyncio functions and contexts

**await** should precede all asyncio operations

# Python async I/O client

```
#!/usr/bin/env python3
import asyncio
from aiohttp import ClientSession

async def fetch(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            return await response.read()

async def bound_fetch(sem, url):
    for idx in range(5): # make 5 attempts
        try:
            async with sem:
                return await fetch(url)
        except:
            await asyncio.sleep(0.1)
```

```
async def run(loop, nclients, nsem):
    url = "http://localhost:8000/?query=test"
    tasks = []
    sem = asyncio.Semaphore(nsem) # Limit # concurrent requests to process
    for i in range(nclients): # number of concurrent clients
        task = asyncio.ensure_future(bound_fetch(sem, url.format(i)))
        tasks.append(task)
    responses = asyncio.gather(*tasks)
    await responses
    print("Responses:", responses)

def main():
    loop = asyncio.get_event_loop()
    # send 10 clients and constrain to process 1k requests
    future = asyncio.ensure_future(run(loop, 10, 1000))
    loop.run_until_complete(future)

if __name__ == '__main__':
    main()
```

**async** should precede all asyncio functions and contexts

**await** should precede all asyncio operations

# Python async I/O server

```
#!/usr/bin/env python3
import json, urllib, random, asyncio, aiohttp
from aiohttp import web

async def worker(params):
    sleep = random.randint(0, params.get('sleep', 10))
    await asyncio.sleep(sleep)
    return msg

async def workflow(params, loop):
    ntasks = random.randint(1, 9)
    tasks = [loop.create_task(worker(params)) for _ in range(0, ntasks)]
    res = await asyncio.gather(*tasks, return_exceptions=True)
    return res

async def process(request):
    params = urllib.parse.parse_qsl(request.query_string)
    loop = asyncio.get_event_loop()
    try:
        task = loop.create_task(workflow(params, loop))
    except asyncio.CancelledError:
        print('Tasks with params=%s has been canceled' % params)
    finally:
        task.cancel()

async def handle(request):
    asyncio.ensure_future(process(request))
    pid = 123 # create and return pid
    body = json.dumps({‘pid':pid}).encode('utf-8')
    ctype = “application/json”
    return web.Response(body, ctype)

def main():
    loop = asyncio.get_event_loop()
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', handle)

    server = loop.create_server(\
            app.make_handler(), '127.0.0.1', 8000)
    loop.run_until_complete(server)
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
    main()
```

# Impact

- If we want to reach top-notch speed for our app we need shift from “sequential” programming to asynchronous/concurrent architecture
- Go language provides this natively, Python via new async module
- Python code will require major code-refactoring
  - identify function/classes participating in asyncio
  - wrap all asyncio operations with async/await prefixes
- Paradigm shift in development
  - asynchronous programming is more complex than classical “sequential” programming
  - event loop; futures & coroutines; synchronisation primitives

# ML R&D: Event classification


- Use available ML frameworks, e.g. scikit-learn, R, Caffe, Theano, TensorFlow, etc.
- Train classifier with multiple physics streams
  - start with supervised learning (traditional ML tools), use few physics channels, get insight into data (feature extractions, transformations, dynamic size of features, clustering)
  - use unsupervised learning (DL networks) to re-discover physics events, find new "signals" and/or anomalies

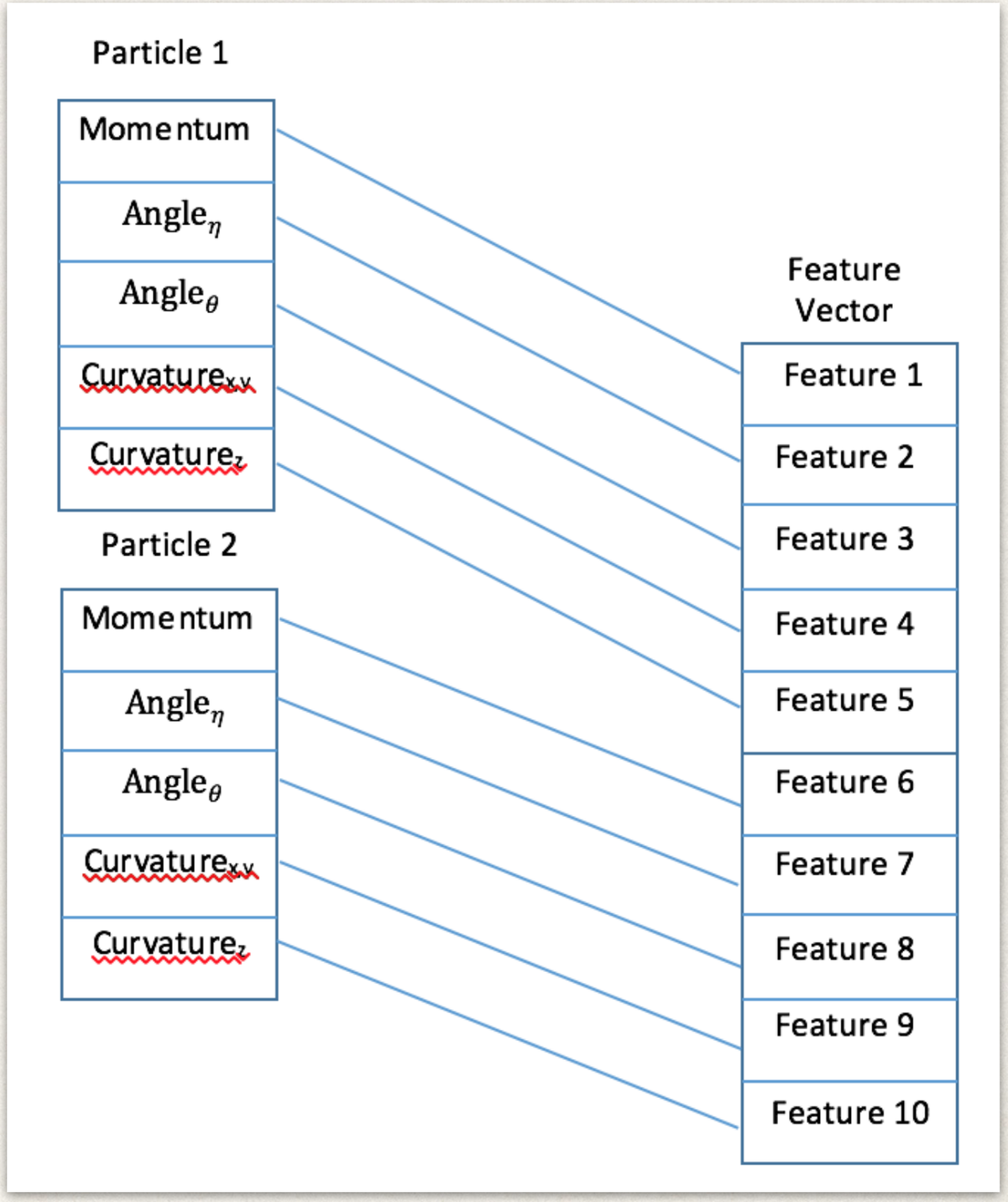
C&O week 2016: http://bit.ly/2noVSua

# Setup

- CMSSW 8_0_21 w/ custom EDAnalyzer
- 4 RelVal GEN-SIM-RECO samples: QCD, JPsi, TTbarLepton, HiggsTaus
- c2numpy to extract track parameters, pixel and silicon hits, see bit.ly/2ehlJ5q , and store them in NumPy arrays
- Scikit-learn libraries for classification

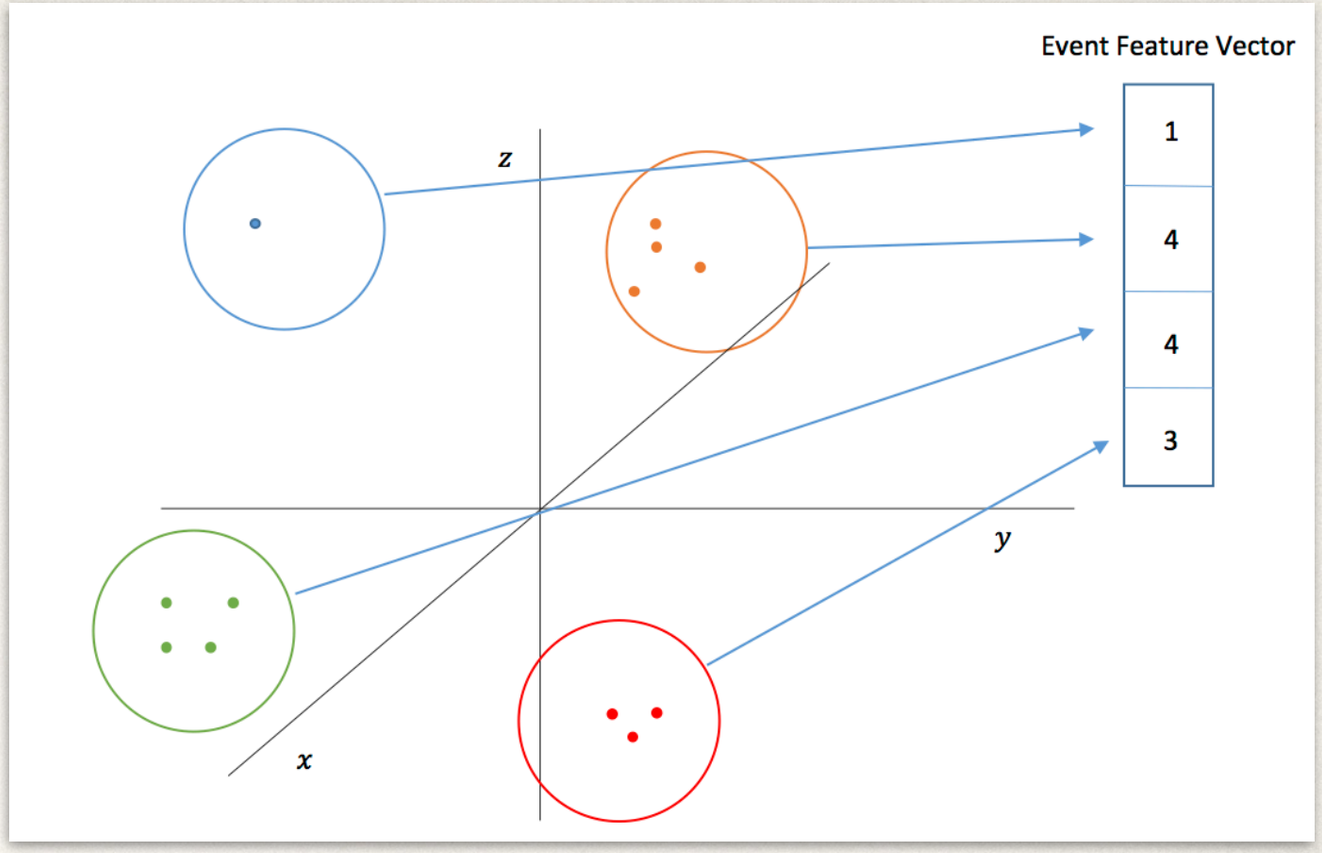
# Concatenation method

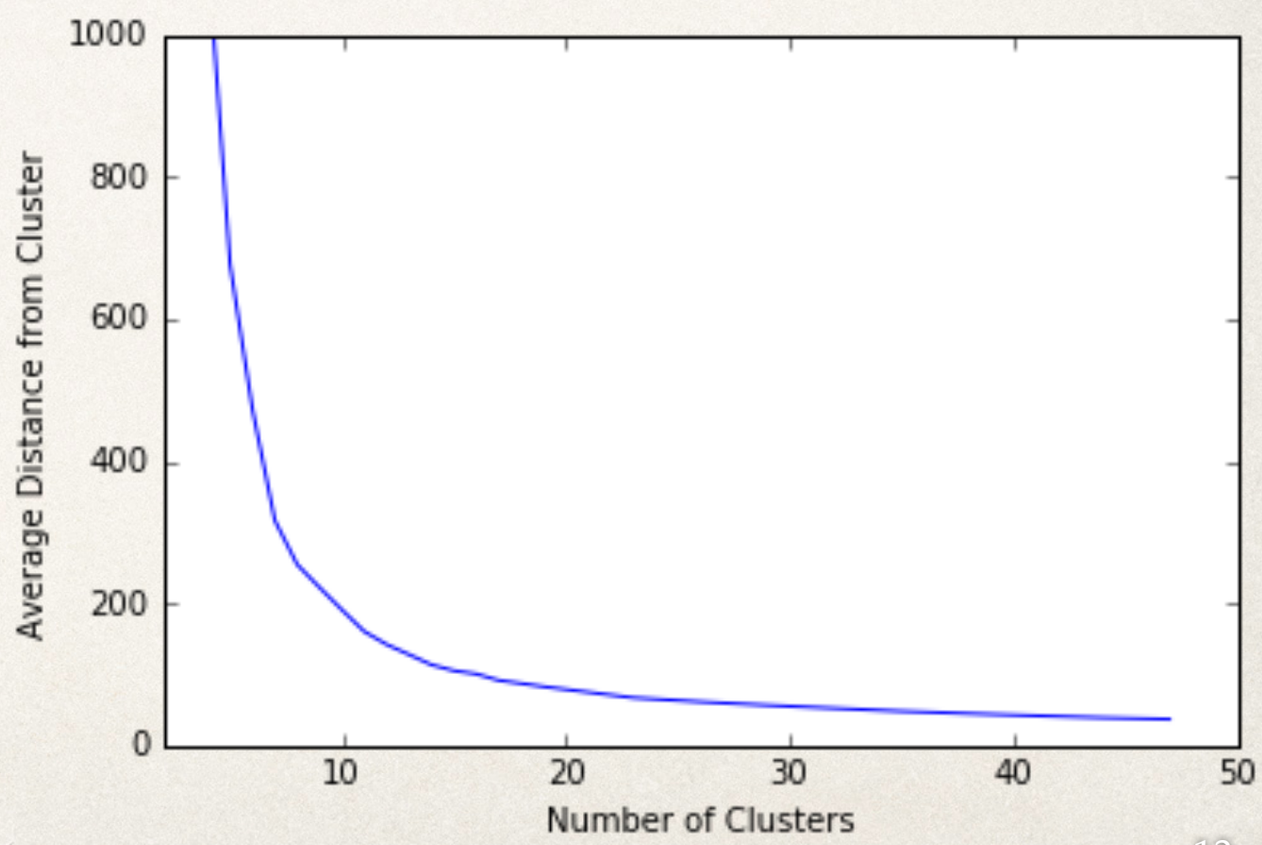
- Sort list of particles in events by their pT
- Take track parameters and concatenate into feature vector, d*k where d is # of features/particle and k is number of particles
- Feature vector is not invariant across events
- One event may have one map to first N-features, while another will have a different map in the same first N-features



# Clustering method

- Perform k-means clustering to assign each particle to a cluster
- In each event find number of particles assigned to each cluster
- Use counts to construct the event feature vector
- Use elbow plot to justify choice of k
- Feature vector is invariant across events



# Preliminary results

| | Filtered concat, 5 features 10 particles | Clustering, 185-features | Clustering, 5-features |
|---|---|---|---|
| Gradient Boosting | 73.9% | 58.7% | 77.6% |
| Random Forest | 75.7% | 57.2% | 72.1% |
| SVM | 76.8% | 69.7% | 71.2% |
| Logistic Regression | 71.9% | 57.6% | 74.2% |

# Preliminary results

| | Overall | Higgs | TTbar | QCD | JPsi |
|---|---|---|---|---|---|
| **Proportion** | 100% | 17% | 14% | 16% | 53% |
| **Gradient Boosting** | 70.7% | 51.0% | 24.8% | 50.5% | 96.0% |
| **Random Forest** | 71.4% | 53.5% | 24.2% | 48.4% | 97.2% |
| **SVM** | 66.5% | 51.5% | 0.6% | 41.9% | 97.0% |
| **Logistic Regression** | 70.6% | 59.2% | 20.0% | 38.7% | 98.0% |

## Clustering algorithm

# Future directions

- Profile individual samples
- DeepLearning on GPU
  - measure accuracy & various benchmarks
- DL/ML on Spark platform via Keras+ML framework
- Need disk space & CPU/GPU cycles for serious work

# Transfer R&D

- PhEDEx is around since 2004
- One of the most successful stories in CMS, but
- Single (central) catalog, ORACLE dependency, no support of user based data
- Explore new ideas, focus on CWP scope and challenges:
  - SD CWP workshop: scalability; leverage new computing and analysis facilities; streaming; http://bit.ly/2npeE4Q

HEP Software Foundation | Activities | Communication | Projects & Support | Get involved! | About

# Google Summer of Code 2017

## Introduction

Google Summer of Code 2017 is a program that allows students to contribute to development of open-source projects, mentored by participating organizations.

Particle physics is an exciting field where large collaborations of scientists collect and analyze the data from high-energy physics experiments, such as the Large Hadron Collider, hosted by the CERN laboratory in Geneva, Switzerland. The questions that we collectively ask are: what are the fundamental blocks that make up our Universe? What is the nature of dark matter and dark energy? What is the nature of the asymmetry between matter and antimatter? What was early Universe like? To answer these questions, particle physicists build software to simulate and analyze what happens in particle physics detectors.

The CERN software for experiments (CERN-SFT) group has participated in the GSoC since 2011. This year the program expands to involve the high-energy physics community under the umbrella of the HEP Software Foundation.

Students interested in participating in the GSoC can find project proposals here.

HSF GSoC Administrators: Sergei Gleyzer sergei@cern.ch and Enric Tejedor Saavedra etejedor@cern.ch

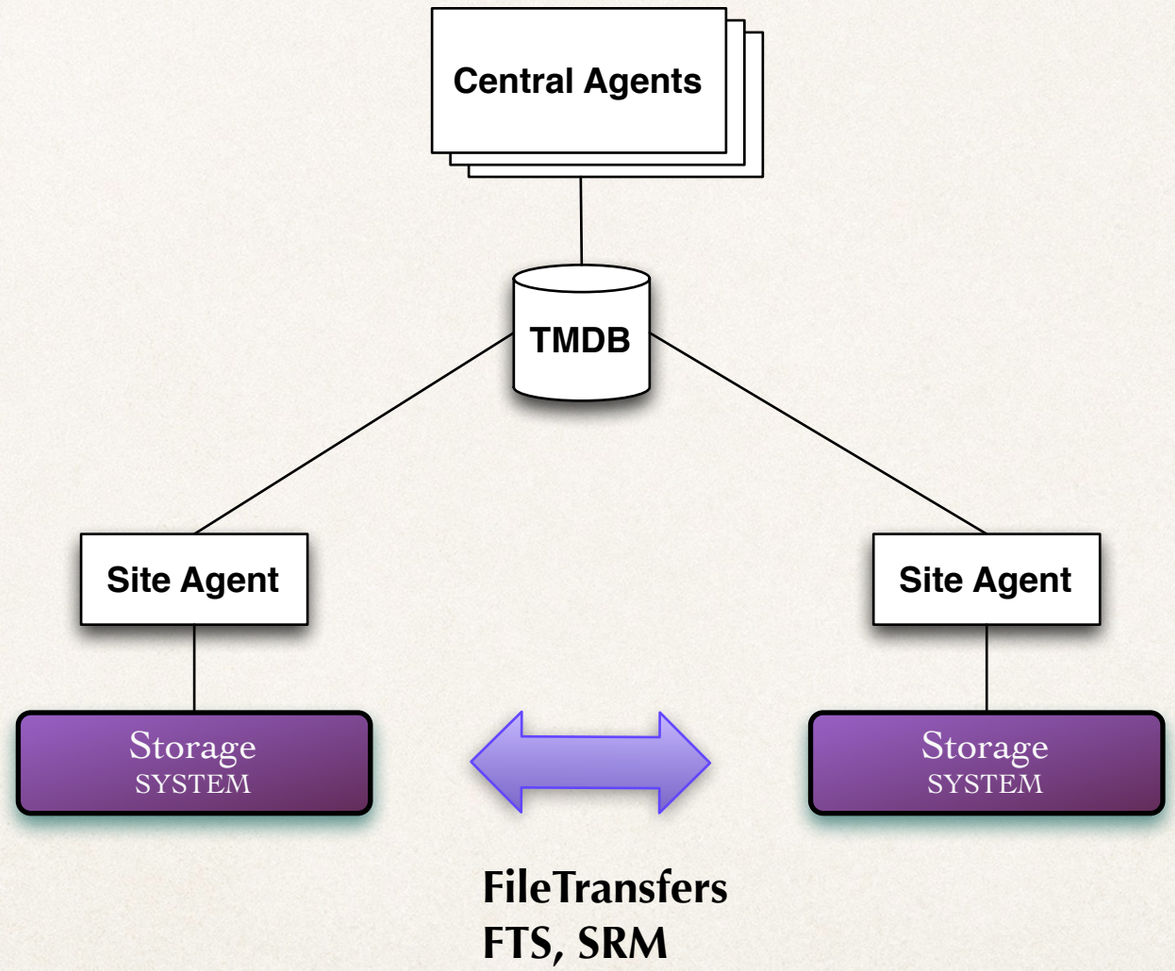
Instructions for participating projects and mentors can be found here.

**http://hepsoftwarefoundation.org/gsoc/proposal_TRANSFER.html**

# Transfer R&D tasks

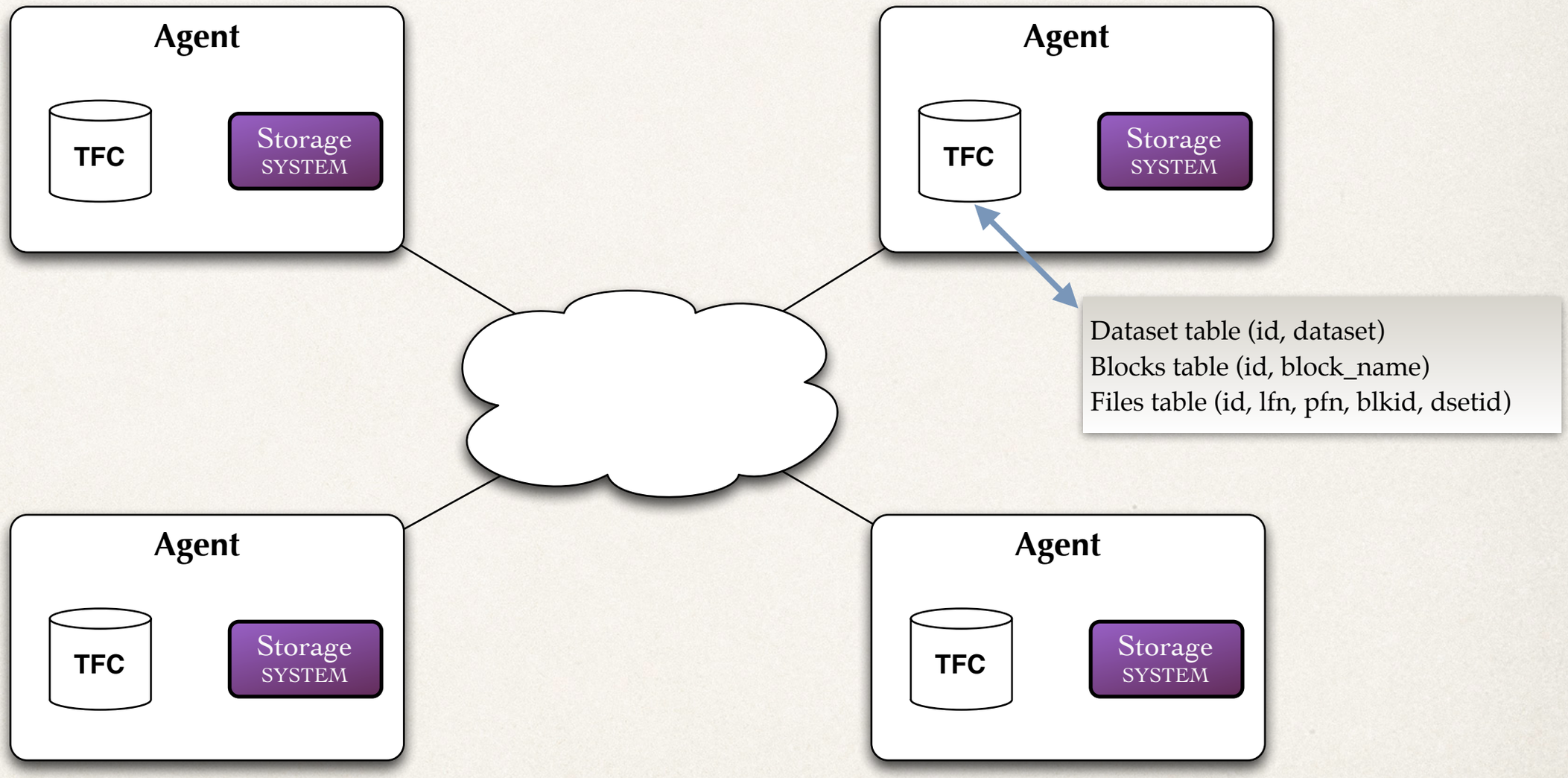
- Eliminate central blackboard system and necessity to rely on ORACLE
- Fully distributed agents with self-discovery and task delegation
- Advanced router capabilities (including ML loopback) & dynamic priorities
- Extend file access to event streaming & data slicing
- Implement support for user based data
- Take advantage of built-in concurrency model of the Go language and explore the scalability boundaries

# PhEDEx architecture




Centralize co-operative transfer tasks, agent daemons

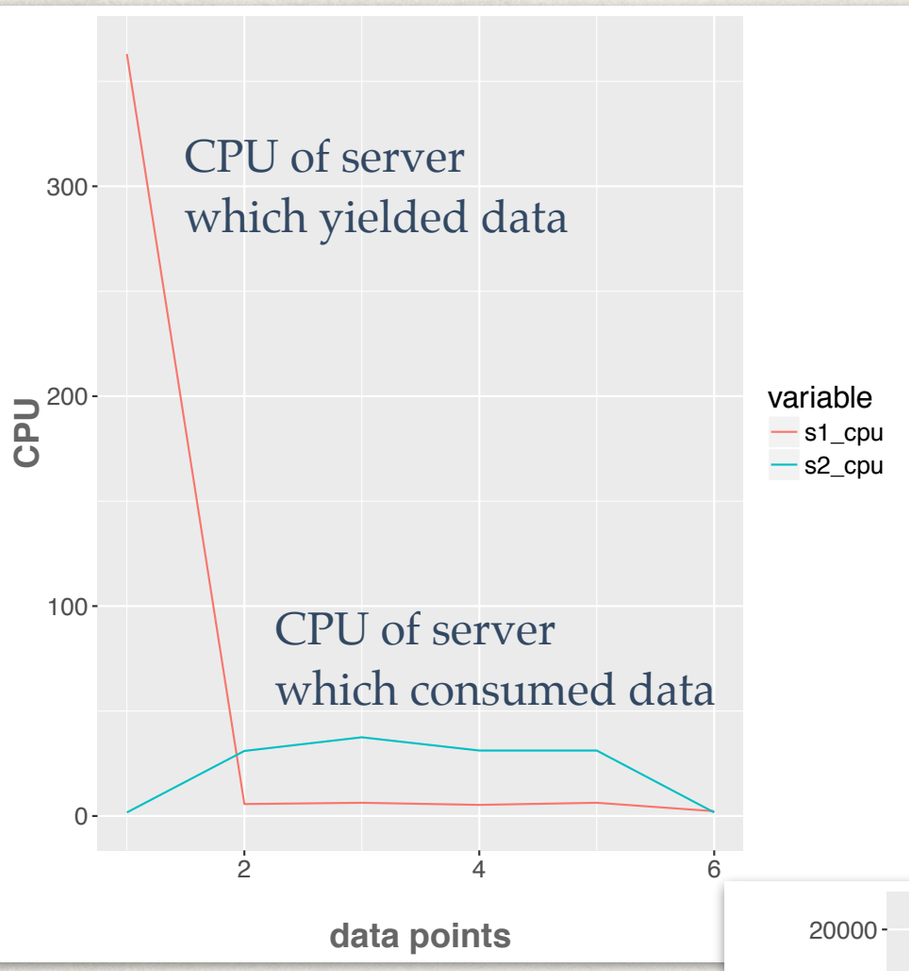
# De-centralize architecture



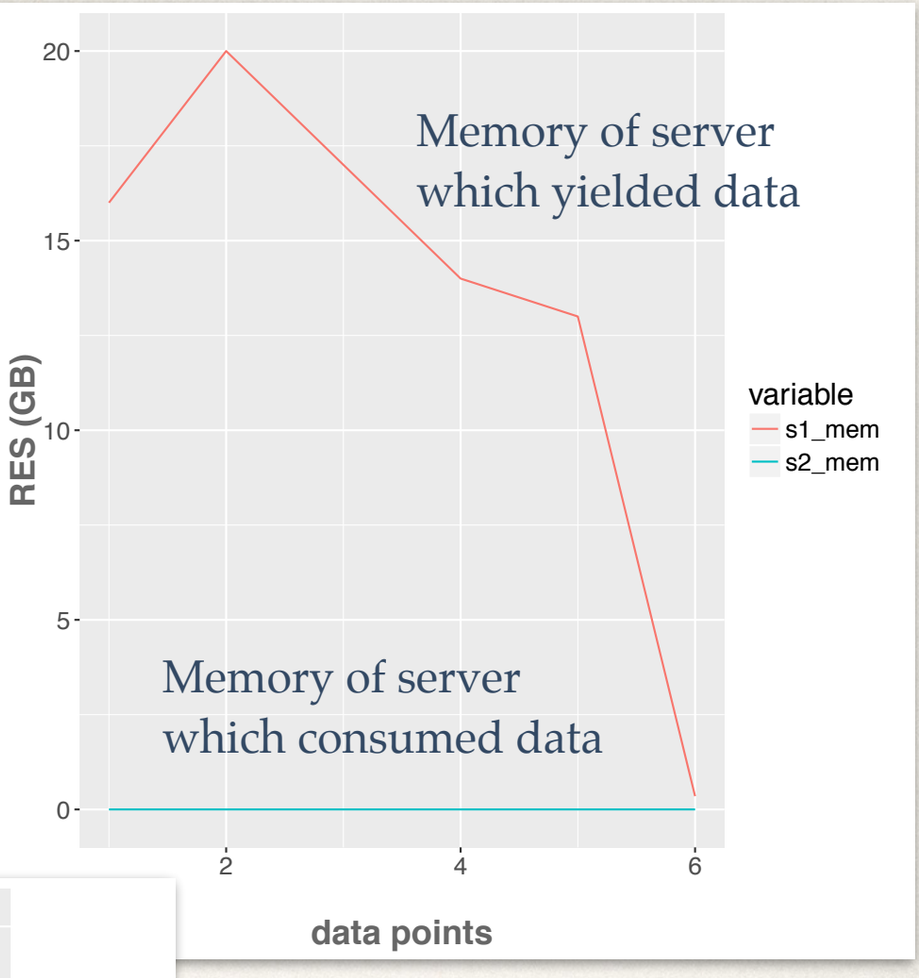

De-centralize agents data-services with auto-discovery and intelligent routing

# Transfer2go for GSoC

- Go-implementation: github.com/vkuznet/transfer2go
- Agents are RESTful data-services, secure HTTP with SiteDB authentication (no need for cmsweb front-end)
- Auto-discovery among agents, requests can be placed to any agent and it will be appropriately re-routed based on up-to-date info from agent routers
- Agent keeps track of transfers and metrics in local DB
- Default transfer protocol is HTTP/FTP, additional protocol/tools can be dynamically added, e.g. FTS
- Zero deployment overhead, i.e. easy to deploy at ANY node regardless of OS version and software stack
- Native concurrency support



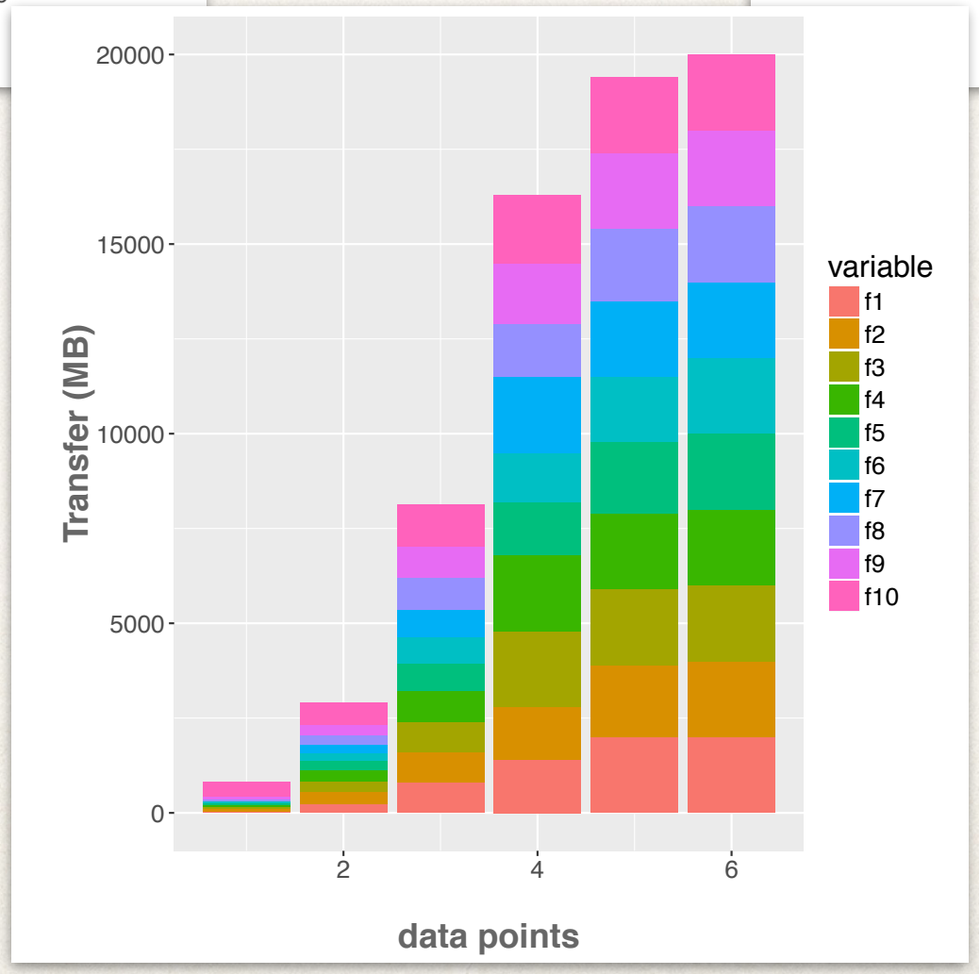

data points
taken from
top command
snapshots
during transfer




Transfer files from
CERN VM to Cornell
via HTTPs protocol

**Server 1** (vocms013):
12 cores, 24 GB of RAM

**Server 2** (lnx231):
24 cores, 32 GB of RAM




**Throughput test:**

Test 1 (10x295MB files)
91Mbit/sec

Test 2 (10x2GB files)
60 Mbit/sec

using 100 Mbit/s
network card