The Disco MapReduce Framework

Chris Mueller ▪ June 2011 ▪ PyGotham

Lab7 Systems, Inc. ▪ Austin, TX
www.lab7.io

MapReduce

MapReduce *is a paradigm for* ***distributed computing*** *developed (patented) by Google for performing analysis on* ***large*** *amounts of data distributed across thousands commodity computers.*

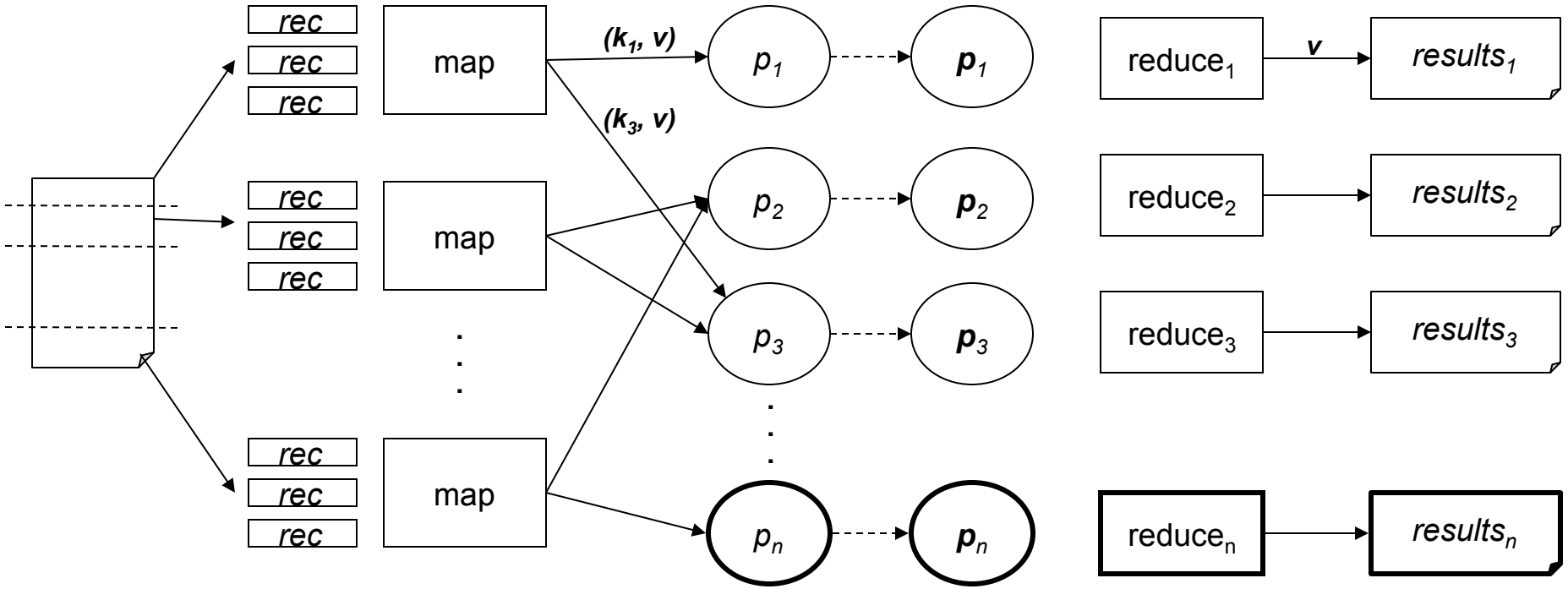
```
Map(rec) -> list(k,v) | Sort(list) | Reduce(k,v) -> list(res)
```

- The **Map** phase processes the input one element at a time and returns a *(key, value)* pair for each element.
- An optional **Partition** step partitions Map results into groups based on a partition function on the key.
- The engine merges partitions and sorts all the map results.
- The merged results are passed to the **Reduce** phase. One or more reduce jobs reduce the *(key, value)* pairs to produce the final results.

MapReduce Dataflow

*The **Map** and **Reduce** operations are inherently parallel and can be dirstributed across nodes of a cluster.*

`Map(rec) | Parition(key) | Sort(p_i) | Reduce(k,v)`



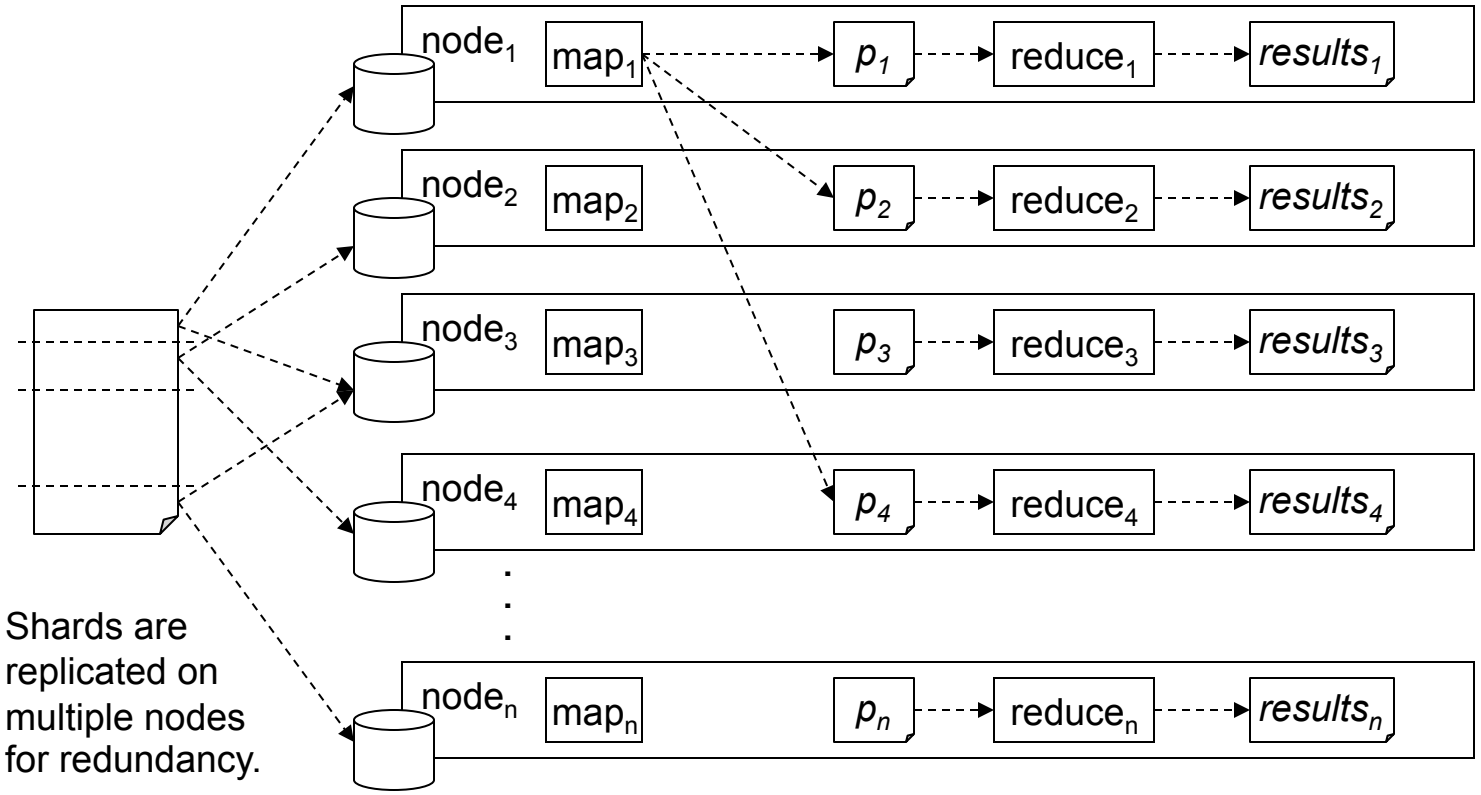
Input files are separated into independent records.

If there are no partitions, only one reduce job is used..

Results are reported for each reduce job.

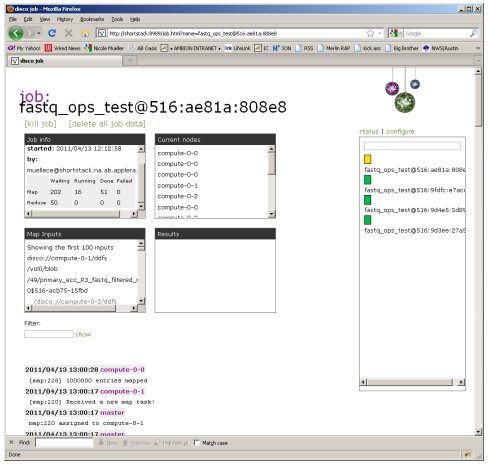
File and Job Distribution

*Most MapReduce engines are designed for use on clusters of **commodity computers** with **large** amounts of **local** storage on each node. Engines typically provide a **distributed file system** for sharding files across the cluster.*



Disco

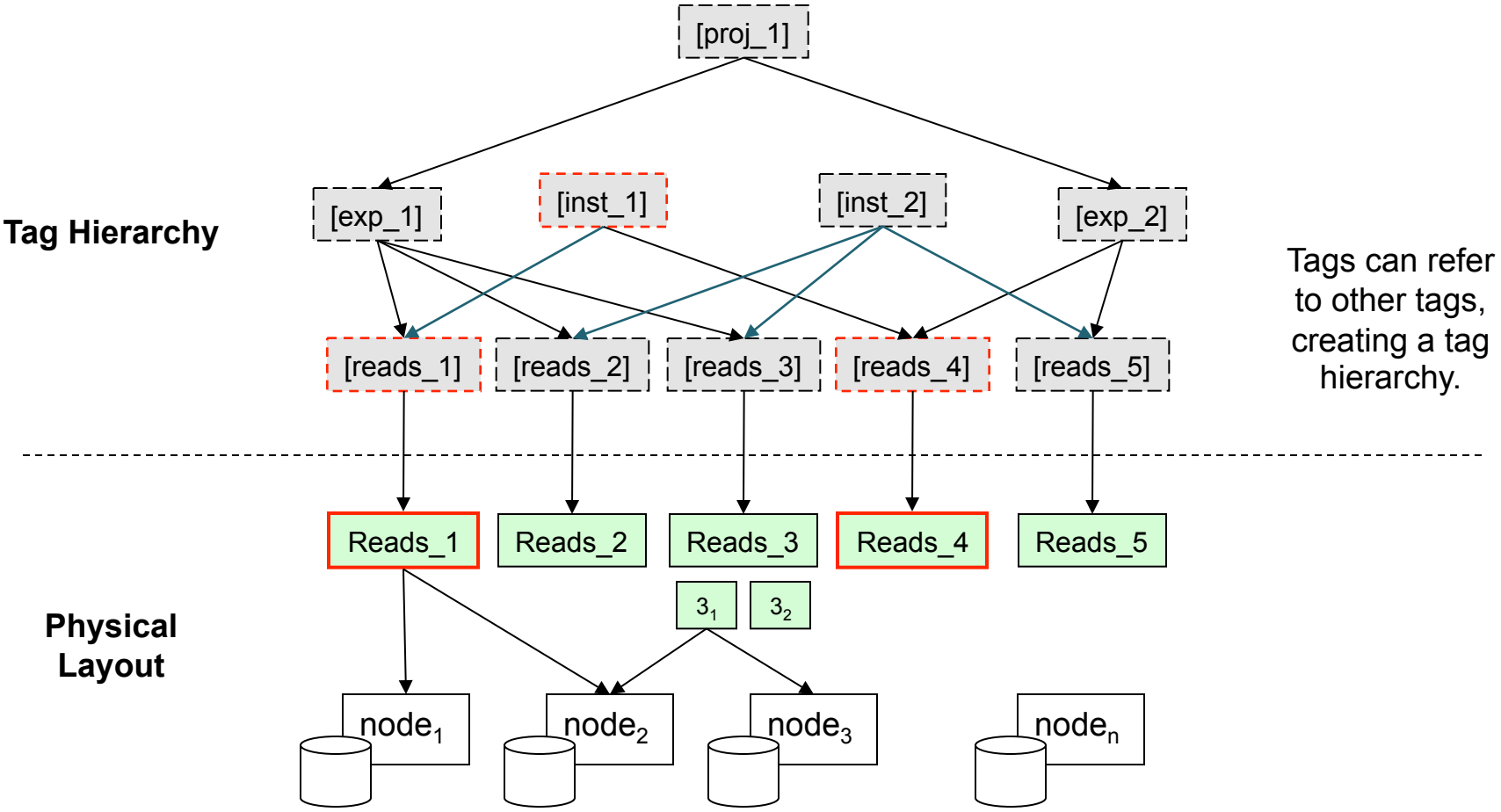
***Disco** is a MapReduce engine built using **Python** and **Erlang**.*



- **Map**, **Partition**, **Reduce**, data processing functions, and CLI are written in **Python**.
- The job distribution engine and basic Web interface are written in **Erlang**.
- File management is handled by the Disco Distributed File System (**DDFS**).

DDFS

***DDFS** is a tag based distributed file system that stores files as complete files or as chunks.*



Example: Loading Data

The DDFS CLI provides a number of commands for interacting with the file system.

```
$ ddfs chunk python:grail:script ./holy-grail.txt
created: disco://compute-0-2/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db
disco://compute-0-0/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db disco://
compute-0-3/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db

$ ddfs blobs python:grail:script
disco://compute-0-2/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db  disco://
compute-0-0/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db
disco://compute-0-3/ddfs/vol0/blob/1f/holy-grail_txt-0$516-af4c8-1a3db

$ ddfs xcat python:grail:script
Sacred-Texts  Legends and Sagas
Note: this is a transcript of the movie produced by an anonymous fan. Obviously the
original is copyrighted and anyone attempting to exploit this file commercially without
permission of Monty Python is a looney...--sacred-texts editor
```

Chunked data is stored using gzip. Non-chuncked data is stored in its native format.

Example: Count Words

***Word counting** is the **Hello World** of MapReduce…*

```
import sys
from disco.core import Disco, result_iterator
from disco.settings import DiscoSettings

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

disco = Disco(DiscoSettings()['DISCO_MASTER'])
results = disco.new_job(
    name="wordcount",
    # input=["python:grail:script"], # DDFS or HTTP load
    input=["http://www.sacred-texts.com/neu/mphg/mphg.htm"],
    map=map,
    reduce=reduce,
    save=True).wait()

for word, count in result_iterator(results):
    print word, count
```

A	28
ALIGN="BOTTOM"2	
ALIGN="CENTER">&quot ;Monty	1
ALL	5
ALL:	8
AM	1
ARTHUR	4
ARTHUR:	194
Aaaaagh!	1
Aaaaugh!	9
Aaauggh.	1
Aaaugh!	4
Aah,	1
Aaugh!	1
Aauuggghhh.	1
Aauuugh.	1
Aauuuves	1

(sorted by count)

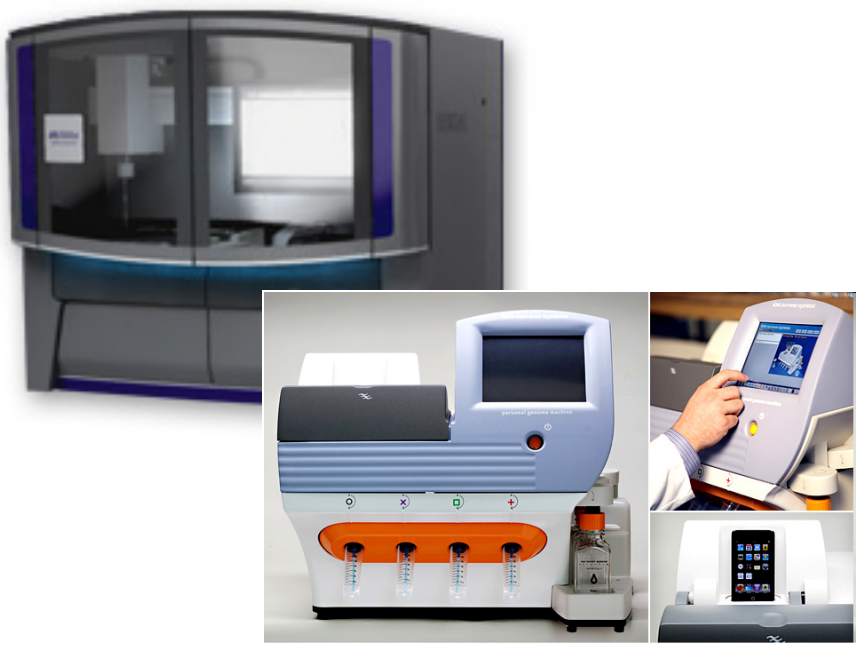
the	289
ARTHUR:	194
I	175
a	164
of	158
you	146
to	138
and	129
is	89
in	88
your	75
Oh,	72
KNIGHT:	68
LAUNCELOT:	65
not	65
FATHER:	57
GUARD	56
GALAHAD:	53

Domain Primer: Next Generation Sequencing

Human Genome Project

10 Years
Thousands of Sequencers
$3,000,000,000
21 Billion Base Pairs (Gbp)

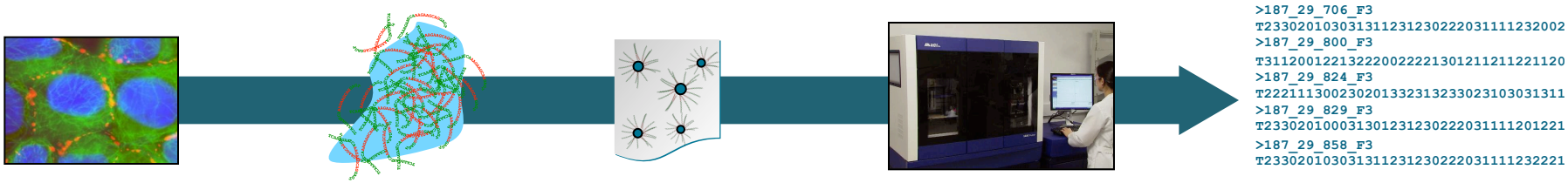
Modern Sequencing



High Throughput	*Benchtop*
2 Weeks	**4 Hours**
One Sequencer	**One Sequencer**
$6,000	**$500**
100-200+ Gbp	**1-100 Gbp**

Next Generation Sequencing

Next generation sequencing (NGS) workflows start in the lab where samples are prepared into "libraries" that are sequenced. The results of a sequencing run are data files containing the sequences and estimates on their quality.



- DNA is fragmented into short segments, 100-200 base pairs long
- During sequencing, each base is read using a "primer" sequence that puts off a color corresponding to the base
- The preparation and sequencing reactions can handle sequences only up to 100 or so bases long
- Each run can produce 100-200 giga-bases of sequencing data from 1-2 billion short reads and generate over a terabyte of raw data
- Once the sequencing is complete, it's up to the informatics staff to make sense of it…

NGS is a BIG Data Problem

The size and complexity of the target species' genome determines how much data is required for a given application.

Human Genomic Applications (3 Gb genome, 20k+ genes)

Application	Coverage	Working Data	SOLiD Runs	PGM** Runs
Assembly	100x (300 Gb)	5-10 TB	3	1,500
Re-sequencing*	15-30x (90 Gb)	300-500 GB	1	450
RNA-Seq*	4-50M reads	5-50 GB	1	1-25
Amplicon*	4-50M reads	5-50 GB	1	1-25

***Multiple barcoded samples are typically used for SNP detection, expression analysis, amplicon sequencing.**
****318 chip**

Instruments are Improving Fast…

- **SOLiD takes two weeks to generate data**
- **PGM takes two hours**
- **Proton will soon generate SOLiD scale data every two hours**
- **Run away now?**

Using Disco to Process Sequence Data

Formatting Records for Map Jobs

*Disco feeds Map one row at a time. Custom **readers** allow Map to operate on any type of record.*

```
@3_41_501_F3
ACCCAGTATTAAGCTTGGTTCGTTCATTCTCTTAACACAATATGTATGAT
+
XVLMgFD<B[NIELT0'(()'''5H552?D4('(A/,+/9/-)'0.###4
```

```
def fastq_input_stream(stream, size, url, params):
  while True:
    seq_id = stream.next().strip()
    seq    = stream.next().strip()
    strand = stream.next().strip()
    qual   = stream.next().strip()
    yield (seq_id, seq, strand, qual)
```

```
@3_41_501_F3    ACCCAGTATTAAGCTTGGTTCGTTCATTCTCTTAACACAATATGTATGAT      +       XVLMgFD<B[NIELT0'(()'''5H552?D4('(A/,+/9/-)'0.###4
@3_41_524_F3    AGCATTAATACCAATACCACTTTTCTCCAGCAACAATCACGCCAGAATAC      +       ggggiRRU]ha^XXX42/,++++?RGFEN^;;BFJ21/2432&#7/*##6
@3_41_682_F3    AATCAATCGTTTGGGCTCATAATAAATTAAAAGTTGCTCCTGCTACTCTG      +       ggggjggegh__]_aC@6422222M3357SFEHJY@?>@@21.-&$$$##
@3_41_748_F3    ATTTTGCTTTCAATTTTTTGGATTGATTATGCTCTCAAACCTTTATTTAT      +       \[\^jggggiZMTUa?<<99/5;DTKJKUgPPMMgB@DJN%%%%821))>
@3_41_789_F3    GTTTCTCCCAGTAGCGTTTCTTCTTTCCATTGTCCTTGTAAGGATGCAAG      +       XVIQgZXMLgIC>BO32*####09A75/4H=82.C-''+2('$#0*%%#.
@3_41_830_F3    TAAACGTATAGAAGAACATAGACATTGGCAAATAGAAACGTTAGGGTTTT      +       ggggkggggjgefgiUUUVVRQRV^LLCDD1%%%3322:;)&&&8$$$#3
@3_41_942_F3    AGCCAATTAATAGATTTATATGAAACTGACTCGAATCAAACTAAAAATTT      +       ggggjb]]eia][bhOFCCC@@@?gXRQZa<=>EU997BC831'<*'''1
```

A typical sequencing run generates about one billion records (~150 GB).

A fastq sequence record spans four lines: id, sequence, direction, quality

The custom input stream reads four lines and returns a tuple.

The stream can be used to load data into DDFS or to read raw fastq files directly into a Mapper.

Partitioning for Reduction

Use a partitioned map reduce to determine the frequencies of each base at each position across all 50-base long sequence reads.

ACCCAGTATTAAGCTTGGTTCGTTCATTCTCTTAACACAATATGTATGAT

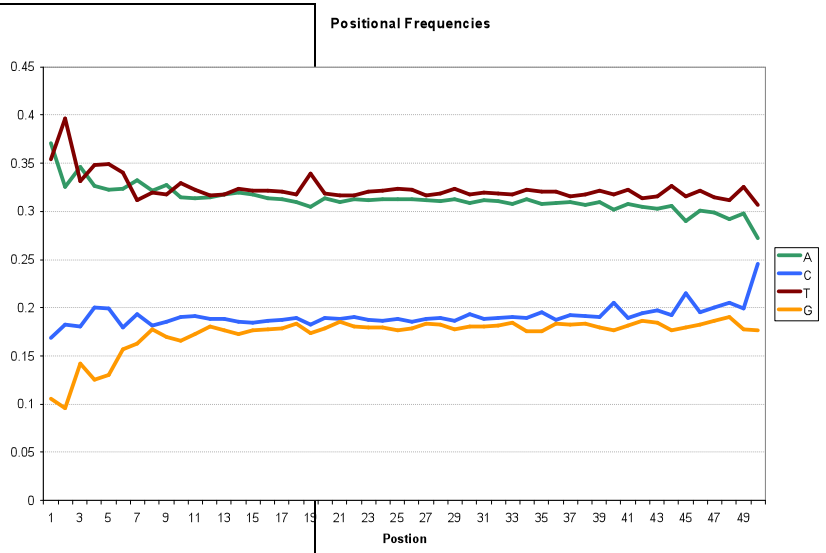
```
def position_map(rec, params):
  for pos, base in enumerate(rec[1]):
    yield (pos, base)

def position_partition(key, np, params):
  return int(key % np)

def position_reduce(iter, out, params):

  counts = {} # pos: {na, nc, ng, nt}
  for pos, base in iter:
    if not counts.has_key(pos):
      counts[pos] = {'A':0, 'C':0, 'G':0, 'T':0, 'N':0}
    counts[pos][base] += 1

  for pos, base in counts.iteritems():
    out.add(pos, base)
  return
```

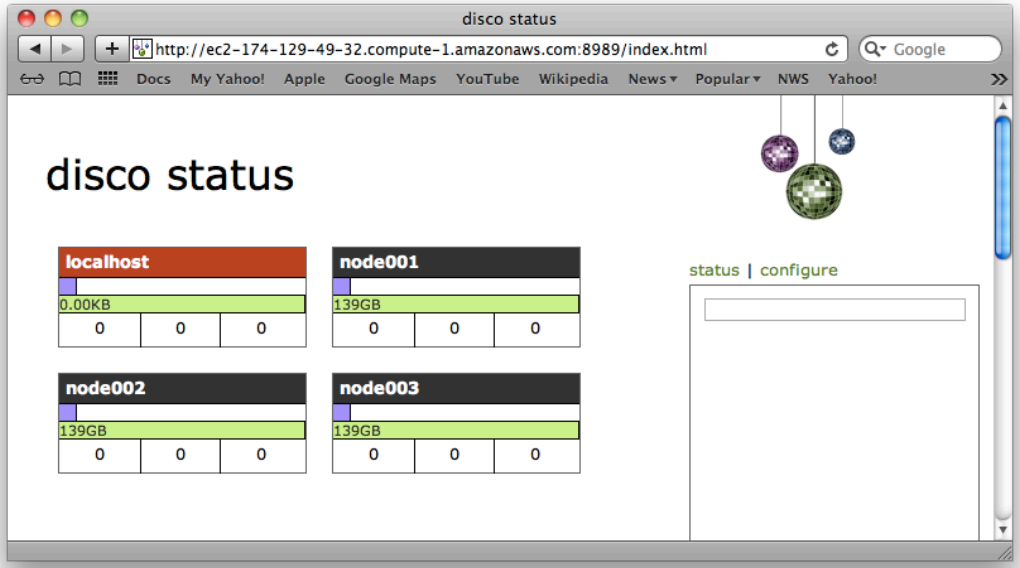


Using Disco in the Cloud

Creating a Disco Cluster with StarCluster

StarCluster *makes it easy to create and deploy clusters on EC2. Of course, StarCluster is a* ***Python*** *application.*

- A pre-built AMI (Amazon Machine Image) has Disco installed
- A StarCluster plug-in configures Disco on each node in the new cluster




```
$ starcluster start -s 4 disco-star-cluster
... cluster startup messages ...
```

https://github.com/cmuellerdev/disco-star

Let's Process Some BIG Data

*The **1000 Genomes Project** is sequencing 1000 individuals. They have **200TB** of sequence data available on Amazon S3.*

`http://aws.amazon.com/1000genomes/`

Let's use our Disco instance to play with the data!

- The data includes BAM (binary alignment files) for each sample that contain sequencing reads that have been mapped to the human genome
- We'll build a pipeline that processes the reads to build a coverage map of the genome, i.e., for each position on the genome, determine how many reads cover that position.

1000 Genomes Coverage Map

```
def read_coverage_map(rec, params):
    ref, read = rec
    yield '%s:%d' % (ref, read.pos), read.qlen

def chr_partition(key, nrp, params):
    key = key.split(':')[0]

    if chr   == 'X':   return 24
    elif chr == 'Y':   return 25
    elif chr == 'MT':  return 0
    else:              return int(chr)

def coverage_reduce(reduce_iter, params):
    import numpy
    chrs = { # Chromosome sizes
        '1':250000000,
        '2':250000000,
        # ...
        'Y':150000000}

    p, l = iter(reduce_iter).next()
    chr, pos = p.split(':')
    c = numpy.zeros(chrs[chr])

    for p, l in reduce_iter:
        chr, pos = p.split(':')
        pos = int(pos); l = int(l)
        c[pos:pos+l] += 1

    yield (chr, ' '.join((str(int(i)) for i in c)))
```

For each read, return the chromosome, position, and alignment length.

Partition the results by chromosome.

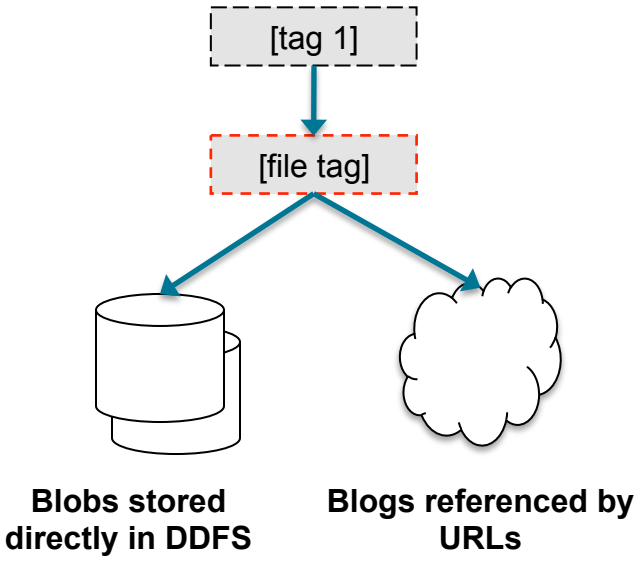
To reduce, create an array containing a count for each location on the chromosome.

Using DDFS and S3

Our cluster uses the local ephemeral storage for DDFS. Each node has approximately 350 GB of storage available, not nearly enough to store all 200 TB of sequence data.

Key DDFS Observations:

- DDFS tags can point to blobs or other tags
- DDFS tags can be URLs



Strategy:

- Create a tag hierarchy where each leaf node is a URL to a file in the 1000 Genomes S3 bucket

Working with Binary Data from S3

The 1000 Genome BAM files binary files stored on S3. We need a way to stream them record-by-record into the Map step. We'll do this by creating a customer reader that caches the S3 files locally.

```
def sam_url_reader(stream, size, url, params):
    import tempfile
    import pysam

    cache = tempfile.NamedTemporaryFile(dir='/mnt')

    block = stream.read(BLOCK_SIZE)
    while block != "":
        cache.write(block)
        block = stream.read(BLOCK_SIZE)

    sam = pysam.Samfile(cache.name)

    for read in sam:
        yield (sam.getrname(read.tid), read)

    sam.close()
    cache.close()

    return
```

`stream` is an open network stream to the S3 file.

Read the data into a temporary file.

Use the pysam library to read records from the BAM file and pass them to the Map phase.

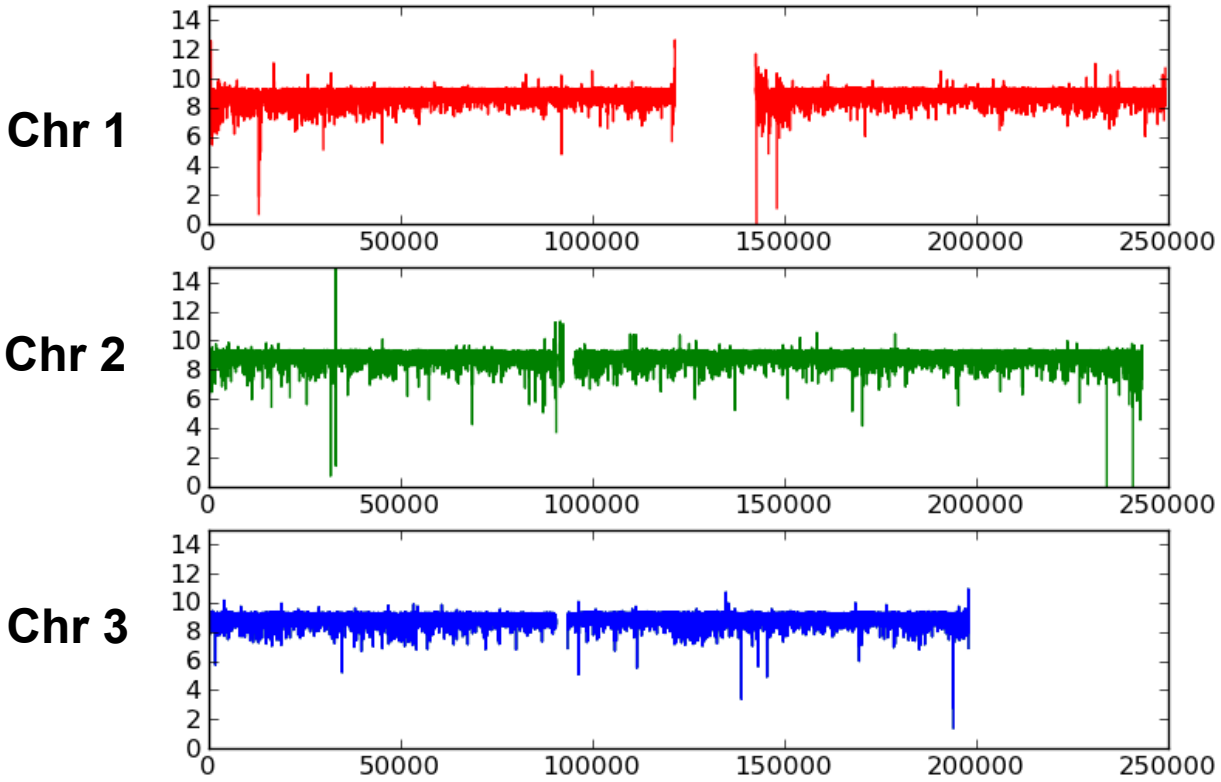
Putting it All Together

Create a Disco job to process all the BAM files. The job will use one map process per BAM file and twenty-seven reduce processes.

```
job = Job().run(
    input = [
      'tag://pop_method:GBR:low_coverage', ...],
    map_reader = sam_url_reader,
    partition = chr_partition,
    partitions = 27,
    map=read_coverage_map,
    reduce=coverage_reduce)

out = open('GBR.low_coverage.out', 'w')
for chr, coverage in result_iterator(job.wait(show=True)):
    out.write('%s %s\n' % (chr, coverage))
```

GBR Low Coverage Genome Map



- Processed 92 samples using a 20 node Disco cluster in 18 hours
- Discussion topic: Is this efficient?

Concluding Thoughts…

- On Disco
  - Disco is a refreshing alternative to the Hadoop hegemony.
  - Disco works well on distributed storage and HPC storage
  - Source is clean and easy to read…
  - …which I discovered while trying to fix installation issues.
- On DDFS
  - The more I use it, the more I like tag-based access to files.
- On MapReduce
  - MapReduce makes solving certain problems on large data trivial.
  - BUT: it's not always a high performance solution. Straight Python, simple batch scheduled Python, and C code can all out perform MapReduce by an order of magnitude or two on a single node for many problems, even for so-called big data problems.
  - Data scale matters: use MapReduce if you truly have large data sets that are difficult to process using simpler solutions.

References

- Jeffrey Dean and Sanjay Ghemawat, *"MapReduce: Simplified Data Processing on Large Clusters".* OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- Disco Project. http://discoproject.org/
- http://en.wikipedia.org/wiki/MapReduce
- http://web.mit.edu/star/cluster/
- http://aws.amazon.com