Computational Methods in Astrophysics

Dr Rob Thacker (AT319E)

thacker@ap.smu.ca

Things to think about

- The investment in google data centres outstrip the largest academic computing centres by orders of magnitude
  - Several million square feet, several million cores, exabytes of disk, running enormous (millions?) numbers of jobs
  - Supposedly, Google processed 24 Petabytes of data *a day* in **2009!**
- At that scale:
  - Things break
    - Programming for fault tolerance is an "If…then" PITA
  - You can't synchronize easily
  - Amdahl's Law is waiting to bite

History

- MapReduce was developed at Google, key paper:
  - "MapReduce: Simplified Data Processing on Large Clusters", by Jeff Dean and Sanjay Ghemawat, 2004
  - It's quite readable – take a look
- The implementation relies on master/worker model
  - Not seen at the user code level
  - Allows fault tolerance to be handled in multiple ways

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and

A note: nothing ever stands still, except MPI ;)

- Apache Mahout is a project to encapsulate a number of machine learning algorithms in a resilient distributed environment
  - Some of them are implemented using Hadoop but Mahout is now more focused on Apache Spark implementations
- Mahout has adopted an R-like syntax for doing linear algebra ("Samsara")
  - Runs on top of Spark (which of course has SparkR!)
- In short, the more people work in this area the more new tools you can expect to see
  - Compare to HPC where APIs were static on 2-3 year timescales
  - That said, all new tools won't be good, and if the web is anything to go by, people will "reinvent the wheel"
  - Hopefully you can pick a platform that works for you!

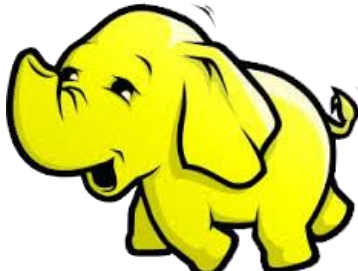
MapReduce and Hadoop

- Let's make a distinction between the environment – Hadoop – and the algorithm, MapReduce
- In essence Hadoop is a storage system combined with an environment that supports certain types of (customizable) batch processing
- There are a number of assumptions about the environment and problem to be addressed
  - The data is too large to be stored in the memory of a single machine
  - You are working on a distributed memory cluster with a distributed file system that is so big failures are inevitable (Hadoop provides the file system)
    - Consequently, you can't rely on specific machine stability! In other words communication must be implicit not explicit

Hadoop – some history

Doug Cutting

- In 2002 Cutting and Cafarella set out to build a new open source search engine, known as Nutch
- Progress is made, but the system is somewhat limited
- Publication of the Google file system (2003) and MapReduce (2004) give new approaches
- They implement in Java compared to C++ at Google
- 2006 Cutting goes to work for Yahoo, the project goes open source
- Project scales out to thousands of nodes & petabytes over 3 years
  - Version 1.0 in 2012, now at 3.3



All you need is MPI? Nope.

- We've discussed this a bit before, but really, take some time to read
  - http://www.dursi.ca/hpc-is-dying-and-mpi-is-killing-it/
  - It's deliberately polemic and provocative
- MPI has enormous flexibility, but with that comes great responsibility
  - It's the parallel programming equivalent of machine code
  - If you get everything right, it flies
  - But there are many potential pitfalls
- MPI does teach you an enormous amount about the complexities of parallelism
  - But could you have spent that time more productively learning other things? That's is an interesting Q.

Simplifying

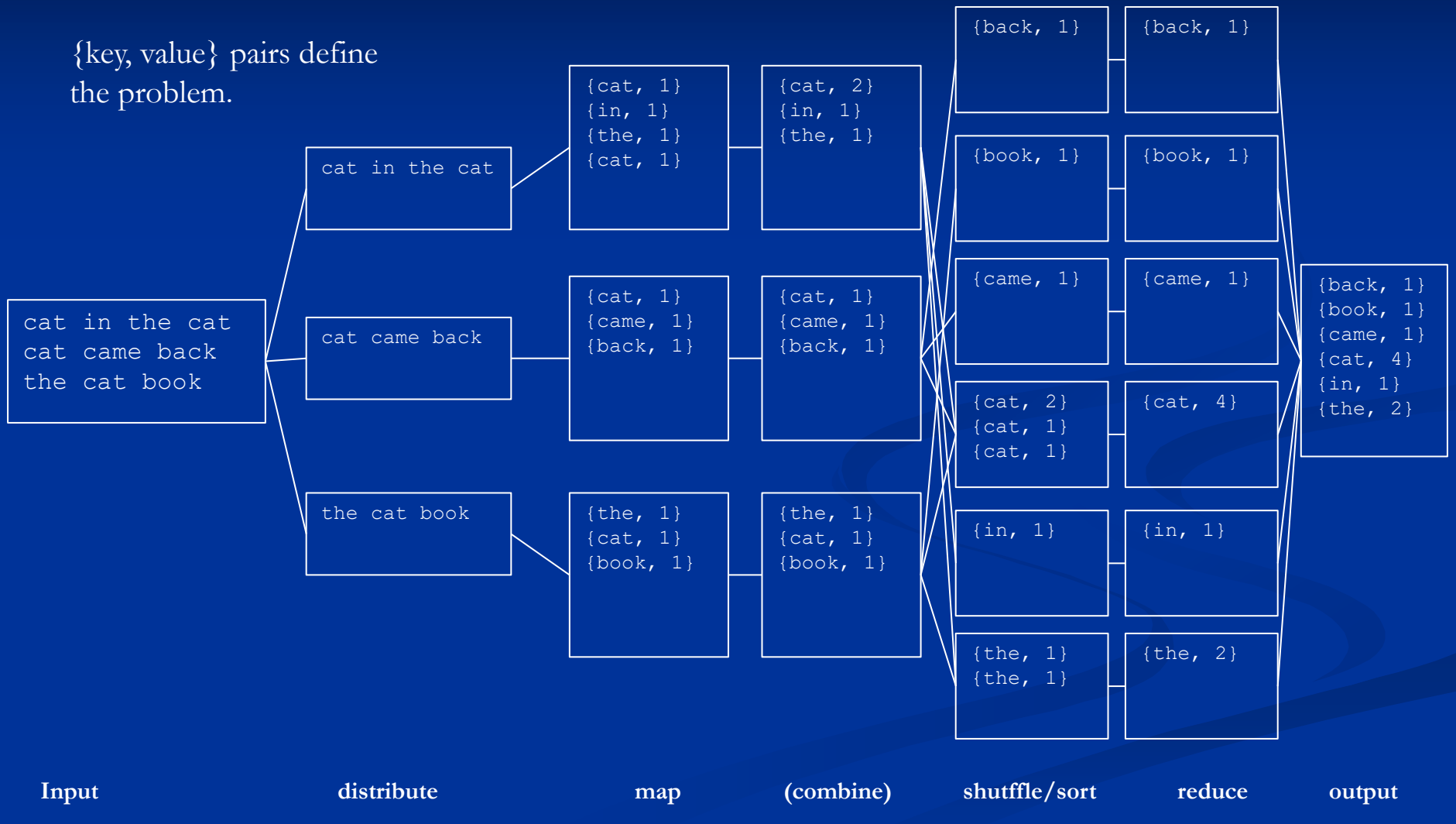
- In the paradigm we've been exploring, the parallel calculation involves
  - Domain decomposition of data
    - In MPI this is incredibly explicit – different domain decompositions require different codes
  - Distribution of the computation
    - This is somewhat less problematic and many programs are naturally written in an SPMD-like form in MPI (but MPMD is possible too)
- Searching, ordering and counting are key steps in many aspects of data analysis
  - So how about creating a framework that can handle algorithms of a certain type?
    - Specifically: ones that involve a mapping (permutation step) and ones that involve a summing (reduction) step
    - It is limiting, but you can work within those limits
  - All the detailed decomposition, node addressing, error checking etc can be handled separately

The canonical example

- Counting the number of appearances of words in a text file
- Steps:
  - 1) Distribute the data
  - 2) Assign keys to the words (mapping step)
  - 3) Count the keys locally
  - 4) Sum the keys globally (usually involves some kind of rearrangement and then a sum)
  - 5) output the answer
- Important point: data come as keys (words) with a total count that corresponds to the sum of appearances
  - So associate a count with each key: pairs {key, count}

Algorithm diagram

{key, value} pairs define the problem.



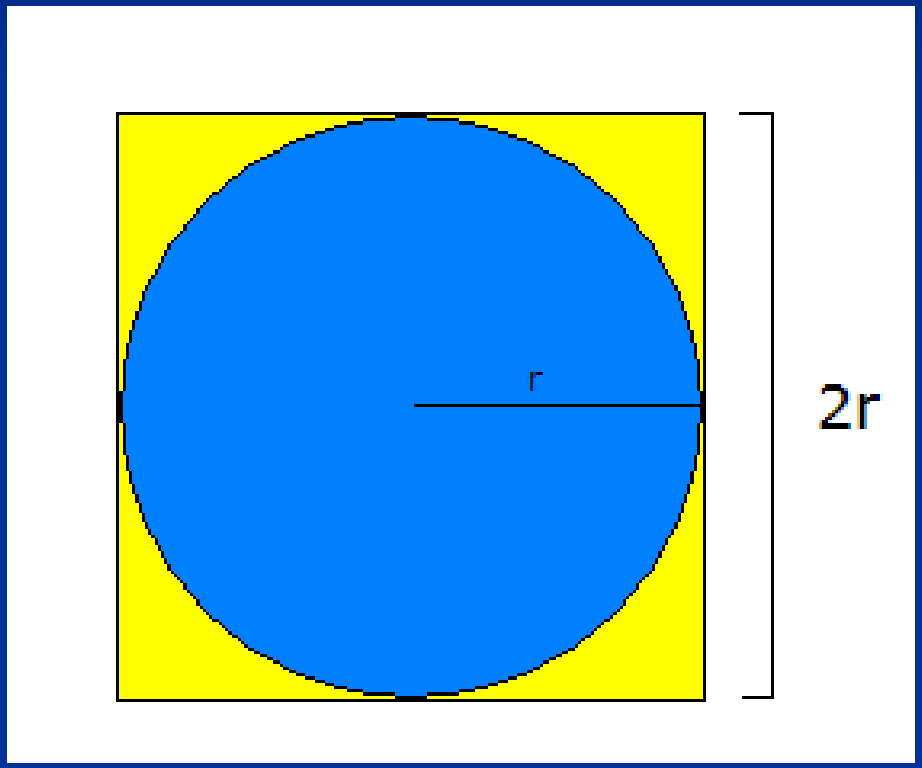
What else can be done in MapReduce framework?

- From Dean and Ghemawat 2004:
  - Distributed grep
  - Count's of URL access frequency
  - Reverse web-link graph
    - i.e. who links to me rather than who do I link to
  - Plenty of others…

What can't be done in MapReduce

- A lot, but that's unfair – it's designed for a specific approach, i.e. problems that can be partitioned and distributed
- Iterative problems (e.g. calculate Fibonacci sequence) obviously won't map
  - That's a general problem for parallelism though
- Algorithms requiring shared global states – there's only one barrier in MapReduce
  - Some Monte Carlo methods fit this
  - This is again an efficiency problem for many parallel APIs

Monte Carlo π



Area of the square:

$A_s = (2r)^2$ or $4r^2$

Area of the circle, denoted

$A_c = \pi r^2$

- $\pi = 4 * A_c / A_s$
- π = 4 * count of pts in the circle / count of points in the square

Monte Carlo π - algorithm

- Randomly generate points on [0,1] then scale to be inside square
- Every task generates P points in square, check which points fall inside circle, count those
- MAP (find r_a = No of pts in circle)
- Need to gather all r_a the count of all pts inside circle
- REDUCE 4*(sum all r_a)/(P * # of tasks)

Parallelised calculation of points on the circle (MAP)

Sum reduction of points then enables calculation of π

Fault tolerance strategy

- In a master/worker model you can
  - Monitor whether workers complete
    - If not spawn the task that failed
  - If master fails you just need to ensure you were storing its state repeatedly
    - Rare but happens
- Reported that google has lost 1600 machines in the process of one job but still had it complete
  - That is *mind blowing* - a single failure will take down most MPI jobs!

Repeated execution vs redundant execution

- How does the system know when something has failed?
  - Maybe one node will take longer than another…
  - Can check that nodes are still up – but that doesn't tell you if there has been a computational kernel crash or something similar
- What about having multiple versions running at the same time?
  - Means you need to increase the size of the computational resources and you will waste cycles
  - But avoids the situation where one node delays the calculation because it takes a long time due to external factors
    - Try repeatedly running a problem that depends on a queue – you'll get different times…
- Additionally – sometimes the inputs are corrupt!
  - If at least two calculations of the same value work go down, you can have the computation continue without them

Coding

- User needs to code the mapper and reducer
- Within Hadoop these can be written in Java, Python, C++
- We'll consider Python examples which are remarkably simple for the word count example
- These Python examples use a little trick
  - Read from stdin and write to stdout
  - Hadoop streaming takes care of ensuring these get piped to the right places

Map & Reduce

- See http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
- There is some confusion over whether a Java jar file needs to be created before running (e.g. using Jython)
  - That is not the case
  - As noted, Hadoop streaming API ensures inputs and outputs are read correctly
- Note this example doesn't have a local sum of words before the sort stage

mapper.py

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

This code needs to be made executable with chmod as well

reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

This code needs to be made executable with chmod as well

Steps to running

- The web tutorial discusses staging data to disk – no need for us to do that here

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar \
-file /home/hduser/mapper.py    -mapper /home/hduser/mapper.py \
-file /home/hduser/reducer.py   -reducer /home/hduser/reducer.py \
-input /user/hduser/gutenberg/* -output /user/hduser/gutenberg-output
```

- That's the execution code!
- Number of tasks is usually determined by the number of file blocks
  - This can be optimized depending upon the nature of the problem

Example output

```
hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar \
-mapper /home/hduser/mapper.py -reducer /home/hduser/reducer.py -input /user/hduser/gutenberg/* \
-output /user/hduser/gutenberg-output
 additionalConfSpec_:null
 null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
 packageJobJar: [/app/hadoop/tmp/hadoop-unjar54543/]
 [] /tmp/streamjob54544.jar tmpDir=null
 [...] INFO mapred.FileInputFormat: Total input paths to process : 7
 [...] INFO streaming.StreamJob: getLocalDirs(): [/app/hadoop/tmp/mapred/local]
 [...] INFO streaming.StreamJob: Running job: job_200803031615_0021
 [...]
 [...] INFO streaming.StreamJob:  map 0%  reduce 0%
 [...] INFO streaming.StreamJob:  map 43%  reduce 0%
 [...] INFO streaming.StreamJob:  map 86%  reduce 0%
 [...] INFO streaming.StreamJob:  map 100%  reduce 0%
 [...] INFO streaming.StreamJob:  map 100%  reduce 33%
 [...] INFO streaming.StreamJob:  map 100%  reduce 70%
 [...] INFO streaming.StreamJob:  map 100%  reduce 77%
 [...] INFO streaming.StreamJob:  map 100%  reduce 100%
 [...] INFO streaming.StreamJob: Job complete: job_200803031615_0021
 [...] INFO streaming.StreamJob: Output: /user/hduser/gutenberg-output
hduser@ubuntu:/usr/local/hadoop$
```

Hadoop Distributed File System (HDFS)

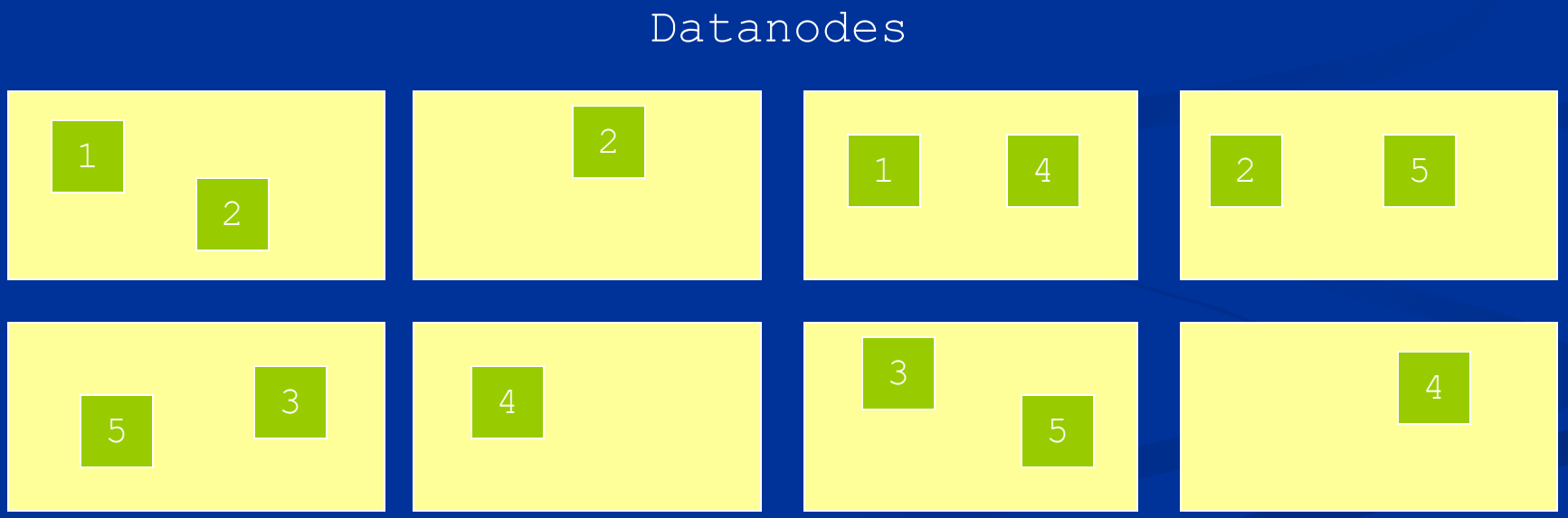
- Appears as a single disk system
  - Runs on top of whatever the native disk system is e.g. ext3
- Daemon based system
- Key part: Namenode
  - Manages file system namespace, metadata and file blocks
  - Often described as running on one machine, but might be several
  - Secondary Namenode
    - Augments primary Namenode by doing an enormous amount of work on the file system edit logs
    - This can be transmitted back to primary Namenode to make restarts easier
    - Some people call the secondary Namenode the “checkpointing node”

Hadoop Distributed File System (HDFS)

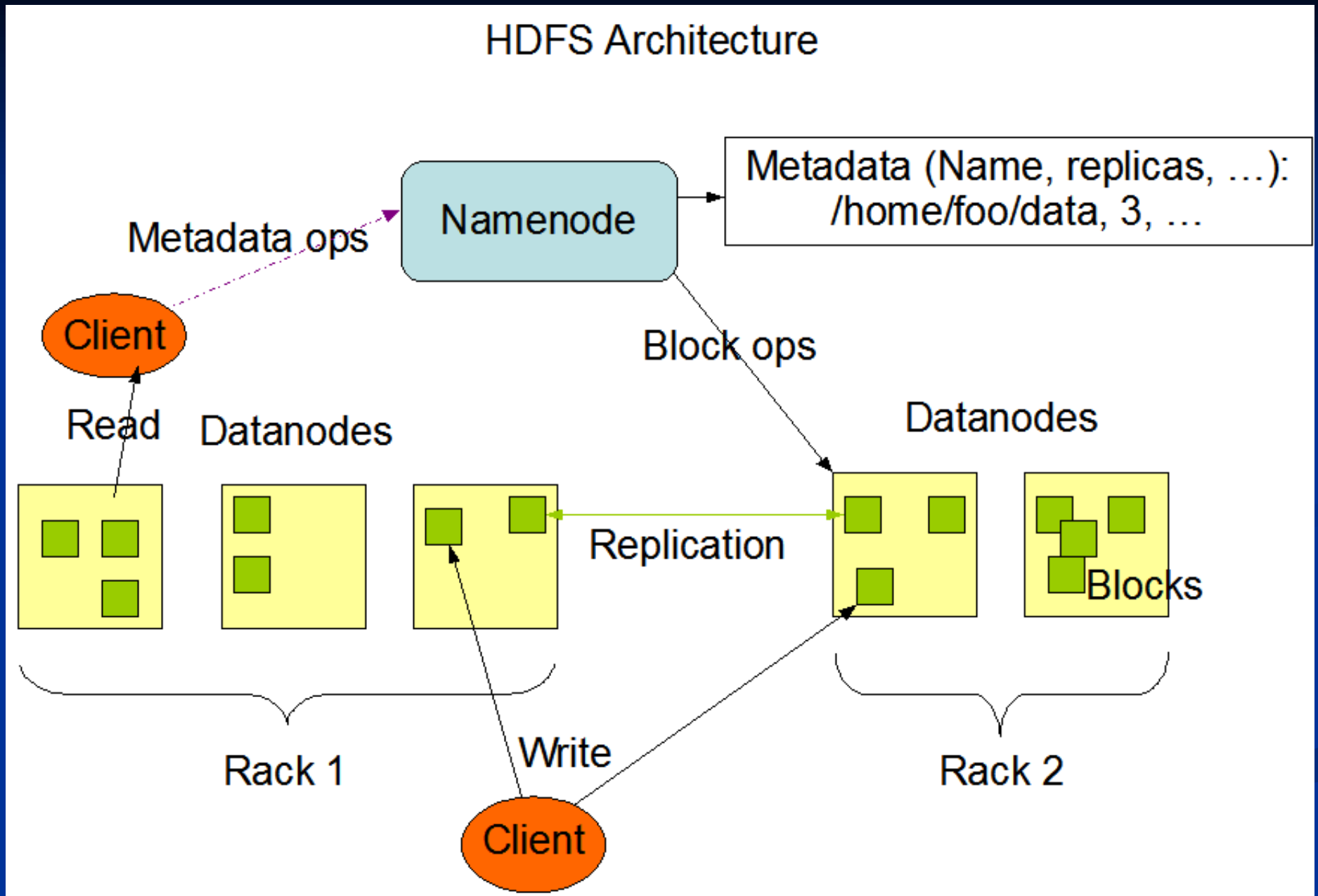
- The NameNode addresses the file which is split into 64 MB blocks and spread (+ replicated) across the DataNodes
  - Files are stored in the regular OS filesystem
  - In practice data is written once and usually read many times
  - Note 64MB is way larger than typical fs blocksizes (e.g. 4k)
  - 1000s of DataNodes are anticipated
- DataNodes regularly inform the NameNode they are operating – "heartbeats", default is every 3 seconds
  - If nothing is received for 10 mins – node is assumed dead

HDFS Data Model

```
NameNode(Filename, replicationFactor, block-ids, …)
/users/user1/data/part-0, copies:2, {1,3}, …
/users/user1/data/part-1, copies:3, {2,4,5}, …
```



Slide by Hairong Kuang, Yahoo!


HDFS Architecture
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

Astro computing goes to the cloud

- While HPC remains wedded to large single machines, Big Data most definitely isn't
- "Infrastructure as a Service" i.e. cloud-based commodity computing is now an industry standard
  - can be used to provision infrastructure on-demand in the cloud on a 'pay as you go' basis
- This is breaking the previous relationship of storing data in a dedicated science archive data centre
  - Although not all projects will choose this kind of abstraction

Astro computing goes to the cloud - benefits

- The key idea is that off-loading management should allow scientists to focus on science
- Data management techniques can effectively be shared
- Smaller projects should be able to re-use methodology from larger ones
- Scaling of compute requirements should be easier
  - In an era of PB data this is a big deal
  - One query might need 1000s of processors
  - The next one tens…
- Number of "white papers" on this for Astro2020

Astro computing goes to the cloud - challenges

- Not all algorithms map well to the cloud
  - Most think true "HPC" facilities will always be needed
- Storage and data movement costs
  - Commercial cloud can be expensive to move data
  - Possibilities for reducing costs via networking platforms though
- Platform "lock-in" and reliance on commercial entities
  - Vendors support specific platforms
  - May not be as big a deal as once thought – rely on easily deployable technologies e.g. Kubernetes

Summary

- MapReduce is conceptually simple in terms of the operations
  - You may have to jump through hoops to make your desired algorithm fit that model (if it can)
- The complexity in Hadoop comes from scaling out to 1000s of machines
  - Fault tolerance through replication, continued updates and a master/worker model
  - Problem sizes are naturally closely linked to the overall size of the input file
- The space of parallel platforms is changing rapidly!
  - And we didn't even really talk about Spark!