Extreme Computing

Introduction to Cloud Computing and MapReduce

Piazza Forum

`https://piazza.com/ed.ac.uk/fall2016/infr11088`

Almost Anything	Piazza
Assignment Questions	Piazza
Extensions	Informatics Teaching Organisation
Harsh Marking	`/dev/null`
Marker Error	The original marker
Appeal Marker Error	Lecturer. Points may go up or down. Include e-mail from marker.
Computer Account	Computing Support

We mark for correctness and efficiency.

Correctly implement the efficient algorithm in:

Python, Java, C++, C, C#, Haskell, OCAML, bash, awk, sed, ...

And run it efficiently $\rightarrow$ full marks.
It does have to run on DICE.

But you made fun of Java?

We'll accept Java.
Just don't complain if it takes you longer to write.

Cluster

We will have a cluster running Hadoop and more.
It's on DICE (the Informatics Linux Environment).

$\Longrightarrow$ No need to install software yourself.
(You can if you want to, but copy your output to the cluster)

Cluster

We will have a cluster running Hadoop and more.
It's on DICE (the Informatics Linux Environment).

$\Longrightarrow$ No need to install software yourself.
(You can if you want to, but copy your output to the cluster)

$\Longrightarrow$ Make sure your DICE account works!
(We don't have `root` so only computing support can help)

Extreme Computing

Introduction to cloud computing, distributed file systems, Hadoop and MapReduce

COMPUTING AS A SERVICE

Google processes 20 PB a day (2008)
crawls 20B web pages a day (2012)

JPMorganChase 150 PB on 50k+ servers
running 15k apps (6/2011)

eBay >10 PB data, 75B DB
calls per day (6/2012)

INTERNET ARCHIVE

>100 PB of user data +
500 TB/day (8/2012)

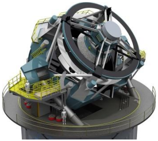
facebook

LHC: ~15 PB a year

CERN

amazon web services™

S3: 449B objects, peak 290k
request/second (7/2011)
1T objects (6/2012)



LSST: 6-10 PB a year
(~2015)

640K ought to be enough
for anybody.

SKA: 0.3 – 1.5 EB
per year (~2020)

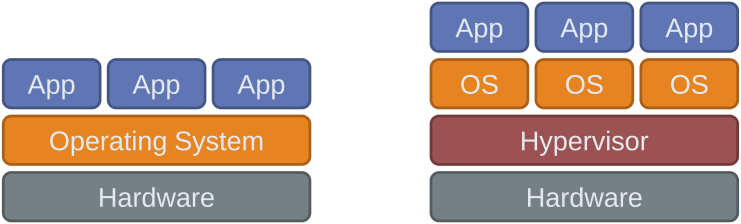
How much data?

Utility computing

- What?
  - Computing resources as a metered service (“pay as you go”)
  - Ability to dynamically provision virtual machines
- Why?
  - Cost: capital vs. operating expenses
  - Scalability: “infinite” capacity
  - Elasticity: scale up or down on demand
- Does it make sense?
  - Benefits to cloud users
  - Business case for cloud providers

THE UNIVERSITY of EDINBURGH
informatics

Enabling technology: virtualisation



Everything as a service

- Utility computing = Infrastructure as a Service (IaaS)
  - Why buy machines when you can rent cycles?
  - Examples: Amazon's EC2, Rackspace
- Platform as a Service (PaaS)
  - Give me nice API and take care of the maintenance, upgrades
  - Example: Google App Engine
- Software as a Service (SaaS)
  - Just run it for me!
  - Example: Gmail, Salesforce

Who cares?

- Ready-made big data problems
  - Social media, user-generated content = big data
  - Examples: Facebook friend suggestions, Google ad placement
  - Business intelligence: gather everything in a data warehouse and run analytics to generate insight
- Utility computing provides:
  - Ability to provision Hadoop clusters on-demand in the cloud
  - lower barrier to entry for tackling big data problems
  - Commoditization and democratization of big data capabilities

So, you want to build a cloud

- Slightly more complicated than hooking up a bunch of machines with an ethernet cable
  - Physical vs. virtual (or logical) resource management
  - Interface?
- A host of issues to be addressed
  - Connectivity, concurrency, replication, fault tolerance, file access, node access, capabilities, services, …
- We'll tackle as many problems as we can
  - The problems are nothing new
  - Solutions have existed for a long time
  - However, it's the first time we have the of applying them all in a single massively accessible infrastructure

Caveats

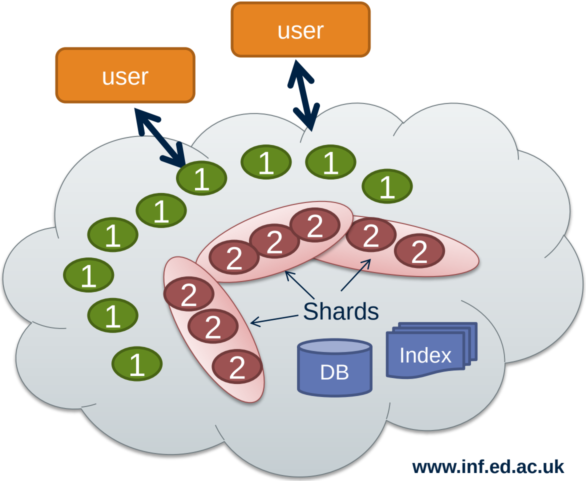
- This is bleeding-edge technology (codeword for immature)
  - We have come a long way since 2007, but still far to go
  - Bugs, undocumented “features”, inexplicable behavior, data loss(!)
  - You will experience all these (those W$*#T@F! moments)
  - When this happens (and it will)
    - Do not get frustrated (take a deep breath)
    - It’s not the end of the world
- Be patient
  - On a long enough timeline everything works
- Be flexible
  - We will have to be creative in workarounds
- Be constructive
  - Tell me how we can make everyone’s experience better

How are cloud structured?

- Clients talk to clouds using web browsers or the web services standards
  - But this only gets us to the outer "skin" of the cloud data center, not the interior
  - Consider Amazon: it can host entire company web sites (like Target.com or Netflix.com), data (AC3), servers (EC2) and even user-provided virtual machines!

Big picture overview

- Client requests are handled in the first tier by
  - PHP or ASP pages
  - Associated logic
- These lightweight services are fast and very nimble
- Much use of caching: the second tier



Many styles of system

- Near the edge of the cloud focus is on vast numbers of clients and rapid response
- Inside we find high volume services that operate in a pipelined manner, asynchronously
- Deep inside the cloud we see a world of virtual computer clusters that are
  - Scheduled to share resources
  - Run applications like MapReduce (Hadoop) are very popular
  - Perform the heavy lifting

In the outer tiers replication is key

- We need to replicate
  - Processing
    - Each client has what seems to be a private, dedicated server (for a little while)
  - Data
    - As much as possible!
    - Server has copies of the data it needs to respond to client requests without any delay at all
  - Control information
    - The entire system is managed in an agreed-upon way by a decentralised cloud management infrastructure

What about the shards?

- The caching components running in tier two are central to the responsiveness of tier-one services
- Basic idea is to always used cached data if at all possible
  - So the inner services (here, a database and a search index stored in a set of files) are shielded from the online load
  - We need to replicate data within our cache to spread loads and provide fault-tolerance
  - But not everything needs to be fully replicated
  - Hence we often use shards with just a few replicas

Sharding used in many ways

- The second tier could be any of a number of caching services:
  - Memcached: a sharable in-memory key-value store
  - Other kinds of Distributed Hash Tables that use key-value APIs
  - Dynamo: A service created by Amazon as a scalable way to represent the shopping cart and similar data
  - BigTable: A very elaborate key-value store created by Google and used not just in tier-two but throughout their “GooglePlex” for sharing information
- Notion of sharding is cross-cutting
  - Most of these systems replicate data to some degree
- We will examine quite a few of these implementations
  - You may have actually used them, do you know how they work?

Do we always need to shard data?

- Imagine a tier-one service running on 100k nodes
  - Can it ever make sense to replicate data on the entire set?
- Yes, if some kinds of information might be so valuable that almost every external request touches it.
  - Must think hard about patterns of data access and use
  - Some information needs to be heavily replicated to offer blindingly fast access on vast numbers of nodes
  - Even if we do not make a dynamic decision about the level of replication required, the principle is similar
  - We want the level of replication to match level of load and the degree to which the data is needed on the critical path

It is not just about updates

- Should also be thinking about patterns that arise when doing reads (aka queries)
  - Some can just be performed by a single representative of a service
  - But others might need the parallelism of having several (or even a huge number) of machines do parts of the work concurrently
- The term sharding is used for data, but here we talk the following
  - Parallel computation on a shard

First-tier parallelism

- Parallelism is vital to speeding up first-tier services
- Key question
  - Request has reached some service instance X
  - Will it be faster
    - For X to just compute the response?
    - Or for X to subdivide the work by asking subservices to do parts of the job?
- Glimpse of an answer
  - Werner Vogels, CTO at Amazon, commented in one talk that many Amazon pages have content from 50 or more parallel subservices that run, in real-time, on the request!

Read vs. write

- Parallelisation works fine, so long as we are reading
- If we break a large read request into multiple read requests for sub-components to be run in parallel, how long do we need to wait?
  - Answer: as long as the slowest read
- How about breaking a large write request?
  - Duh… we still wait till the slowest write finishes
- But what if these are not sub-components, but alternative copies of the same resource?
  - Also known as replicas
  - We wait the same time, but when do we make the individual writes visible?

Replication solves one problem but introduces another

More on updating replicas in parallel

- Several issues now arise
  - Are all the replicas applying updates in the same order?
    - Might not matter unless the same data item is being changed
    - But then clearly we do need some agreement on order
  - What if the leader replies to the end user but then crashes and it turns out that the updates were lost in the network?
    - Data centre networks are surprisingly lossy at times
    - Also, bursts of updates can queue up
- Such issues result in inconsistency

Eric Brewer's CAP theorem

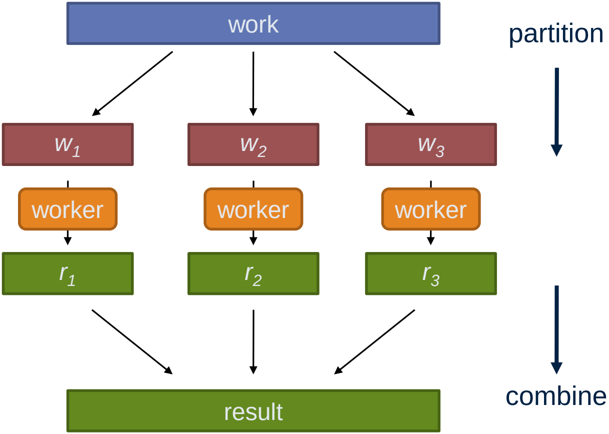
- In a famous 2000 keynote talk at ACM PODC, Eric Brewer proposed that
  - "*You can have just two from Consistency, Availability and Partition Tolerance*"
- He argues that data centres need very fast response, hence availability is paramount
- And they should be responsive even if a transient fault makes it hard to reach some service
- So they should use cached data to respond faster even if the cached entry cannot be validated and might be stale!
- Conclusion: weaken consistency for faster response
- We will revisit this as we go along

Is inconsistency a bad thing?

- How much consistency is really needed in the first tier of the cloud?
  - Think about YouTube videos. Would consistency be an issue here?
  - What about the Amazon “number of units available” counters. Will people notice if those are a bit off?
    - Probably not unless you are buying the last unit
    - End even then, you might be inclined to say “oh, bad luck”

SETTING UP WORKFLOWS

Key premise: divide and conquer



Parallelisation challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What's the common theme of all of these problems?

Common theme?

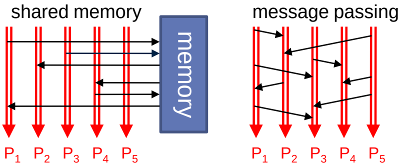
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

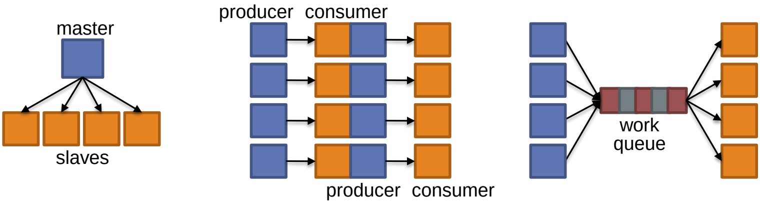
Managing multiple workers

- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know when workers need to communicate partial results
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

Current tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues





Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything
  - The MapReduce runtime alleviates this

What's the point?

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the what from how
  - Developer specifies the computation that needs to be performed
  - Execution framework (aka runtime) handles actual execution

The data centre *is* the computer!

Source: Google

MAPREDUCE AND HDFS

Big data needs big ideas

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster has limited bandwidth, cannot waste it shipping data around
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable, memory throughput is even better
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour
- Computation is still big
  - But if efficiently scheduled and executed to solve bigger problems we can throw more hardware at the problem and use the same code
  - Remember, the datacentre is the computer

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

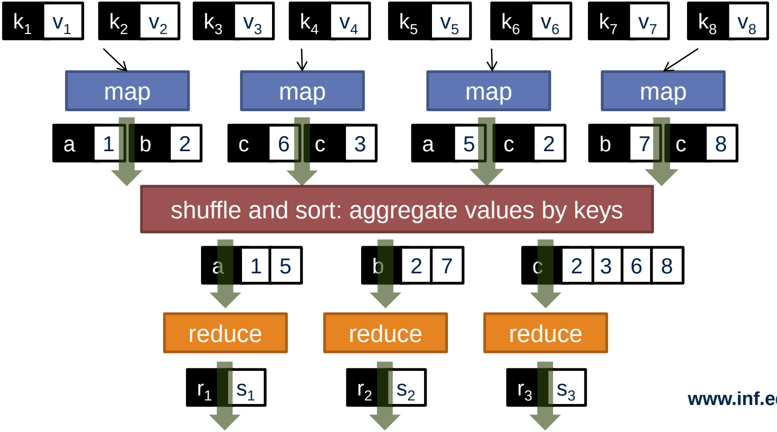
MapReduce

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [<k_2, v_2>]$

  reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

  – All values with the same key are sent to the same reducer



MapReduce runtime

- Orchestration of the distributed computation
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles data distribution
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (more information later)

MapReduce

- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’) → <k’, v’>*

  – All values with the same key are reduced together
- The execution framework handles everything else
- This is the minimal set of information to provide
- Usually, programmers also specify:

  partition (k’, number of partitions) → partition for k’

  – Often a simple hash of the key, e.g., hash(k’) mod n

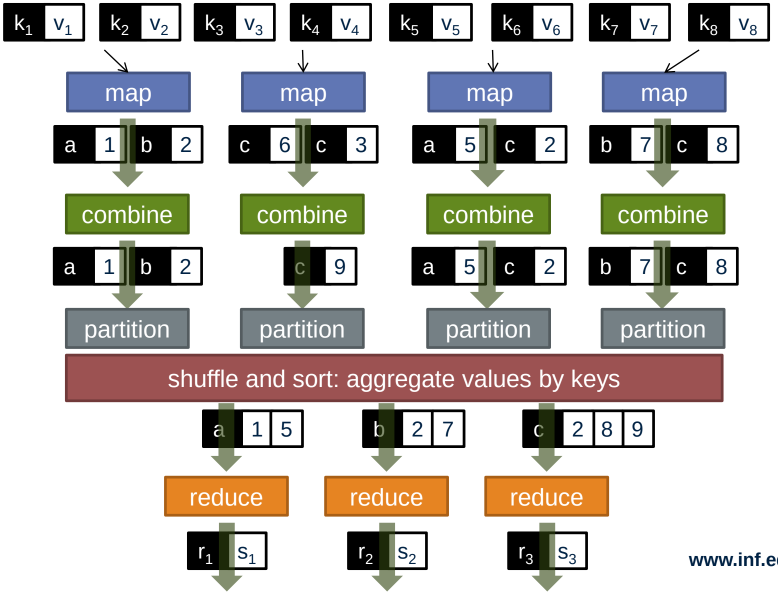
  – Divides up key space for parallel reduce operations

  combine (k’, v’) → <k’, v’>*

  – Mini-reducers that run in memory after the map phase

  – Used as an optimization to reduce network traffic

Putting it all together



Two more details

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

“Hello World”: Word Count

```
Map(String docid, String text):
     for each word w in text:
          Emit(w, 1);

Reduce(String term, Iterator<Int> values):
     int sum = 0;
     for each v in values:
          sum += v;
     Emit(term, sum);
```

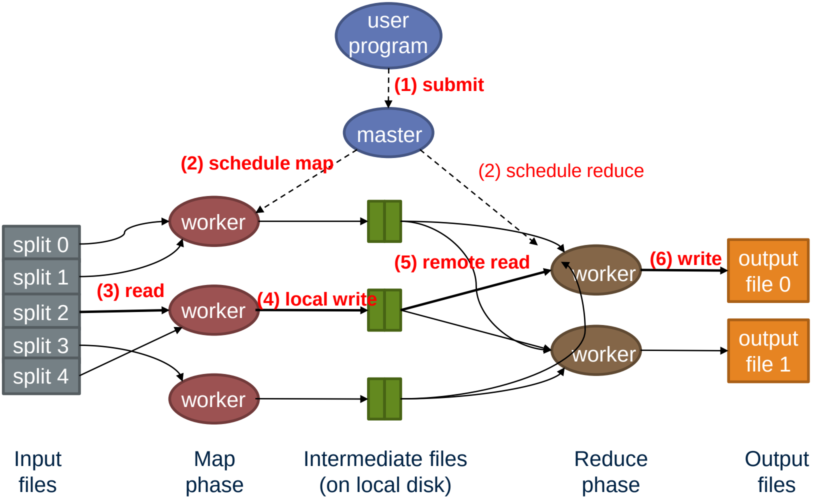
MapReduce can refer to…

- The programming model
- The execution framework (aka runtime)
- The specific implementation

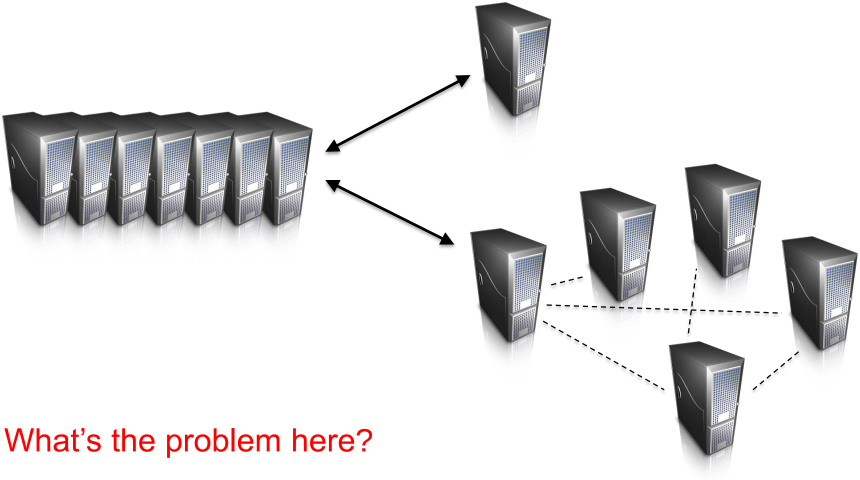
Usage is usually clear from context!

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.


user program
(1) submit
master
(2) schedule map
(2) schedule reduce
split 0
split 1
split 2
split 3
split 4
(3) read
worker
worker
worker
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disk)
Reduce phase
Output files

How do we get data to the workers?



What's the problem here?

Distributed file system

- Do not move data to workers, but move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over exotic hardware
  - Scale out, not up
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

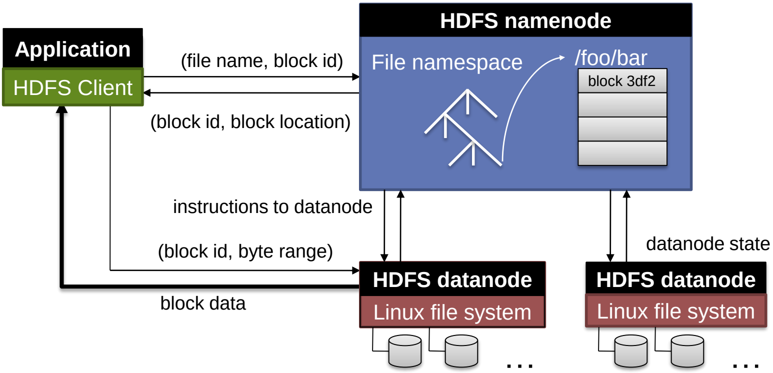
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Different consistency model for file appends
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology

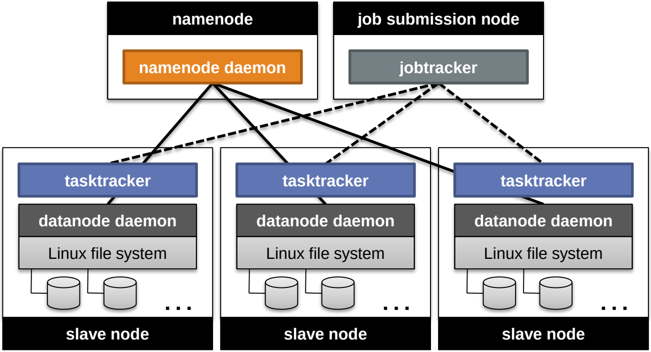
HDFS architecture



Namenode responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together



Summary

- Introduced the notion of utility computing
- Introduced cloud computing and the need for infrastructure
- Presented some of the tools necessary for manipulating Big Data
- We will next turn to the internals of such platforms