Chapter

Disk Access Options

1 Storage Devices

Single core need performance from disks to keep the pipeline full.

Multi-core need the performance to keep all the pipelines full.

Single core has essentially one route to the ALU, starting from the disk and ending at the register, but the first part of the route from disk is slow.
Multi-path.

"random" access for the programme. Data is often read sequentially so "pre-fetch" can be used to mitigate stalls

One possible solution is to create multiple routes off the disk. Another is to use a fast disk (SSD) for the programme and a slower larger disk for the data

Data duplication by hand is too time consuming and unreliable.

Many cores mean we have a route ending in the core for each core. If we duplicate the data we can provide a completely different route for the data, but we have to control data duplication.

Raid

Providing alternate data paths in both instances means providing more than one source for the data.

Raid for single processors and data duplication and distribution for multi-core.

First look at how raid works.

Data duplication and distribution Hadoop
(not talking about use but about the underlying technology)

Summary

Dependability is vital

Suitable measures

Latency – how long to the first bit arrives

Bandwidth/throughput – how fast does stuff come through after the latency period

Obvious corollary – bandwidth for small block is lower than large blocks. Manufacturers quote favourable numbers (for as long as they can get away with it).

Desktops performance dominated by response time
Servers by throughput.

Multimedia processing on the desktop may need excellent throughput

Dependability

Fault is the failure of a component
Component failure does not necessary lead to system failure.
System failure leads to **service interruption**.
Restoration is return to **service accomplishment**

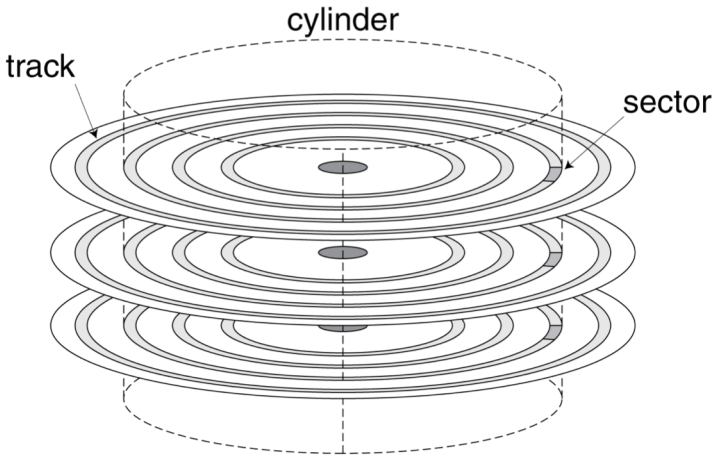
Whirlwind was said to provide 35 hours a week with 90%
Which in 1951 was excellent

Raid

Calculating dependability

Manufacturers quote *mean time to failure.* MTTF
depends on device
Service interruption *mean time to repair* MTTR
depends on your organisation
Mean time between failures MTBF = MTTF + MTTR
Availability = MTTF/(MTTF+MTTR)

Availability is something you can manage.
If MTTR = 0. Availability can be 100% even with a small MMTF.
Rapid replacement may trump high dependability; and be cheaper. *Remember to include staff costs*

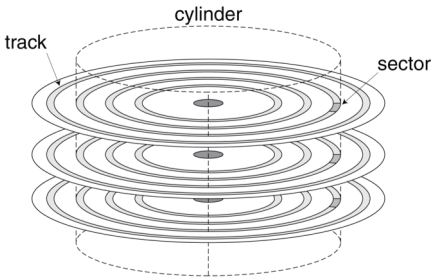


Access

Sector: Sector ID. Data – fixed number of bytes eg 512
Error correcting code, synchronization field and gaps.
ECC on controller – system may never see errors.
Remapping sectors, forward error correction. Need to monitor for errors to replace before failure.

Access:
Queuing delay if other accesses pending. On board control may re-arrange accesses "elevator fashion".
Again conflict between use "as fast as possible" & service "as much as possible"
Seek – head movement; all heads together, hence cylinders. Arrange data in cylinders for fast retrieval
Rotational latency
Data transfer
Controller overhead – (error correction)

Actual seek may be better than average – locality
Drives may include cache memory, which acts in a similar way to CPU cache. Fetching more data than required. Pre-fetching subsequent blocks.



Clever controllers make it hard to predict what will happen.
How does CPU pre-fetching interact with the disk

IO commands

Devices managed by special purpose hardware
- Transfers data
- Synchronises with software

Command registers – control the device
Status registers – information including errors
Data registers – to transfer data to and from the device

Instructions can often only be accessed in kernel mode
Memory mapped I/O. Registers are in the same space as memory.
Address decoder distinguishes between them.
Simplifies programming – hard work delegated to the OS

Device ready (or error) CPU interrupted. Must handle
Can priority order devices, so one device may interrupt another.
High speed transfer using *direct memory access DMA.*
OS provides starting address and controller working autonomously transfers data. Interrupt on completion.

Very useful for large transfers

Measure performance

Make sure you are measuring what you think you are:
Disk controllers may re-order requests and pre-fetch results. The bus used has an influence.
The OS you are running. The application you are using.
DBMS tend to do clever things with disks – the results for a database may not reflect a simpler app.

Response time and throughput are antagonistic.

Transaction Processing Council (TPC) develop benchmarks to estimate database performance.
Small amounts of data transferred (DBMS queries).
Measure I/O rate, not data rate – may count overheads as part of the payload and give a larger number than you will see.
TPC benchmarks published and used by industry.
Hardware/Software suppliers may tune their systems to score well in TPC (true of all benchmarks)
Also benchmarks for file systems, by stressing an NFS server with a workload based on actual measurements.
Benchmarks for web servers.

Choose benchmarks to match application. Beware of tuning

Why arrays?

Users need space to store data for their work.
Simplest is to provide them with a disk.
When they run out of space provide them with a larger disk.
When they have filled the largest disk provide them with a second disk.
Repeat ….

Drawbacks

what happens to the smaller disks.

How does the user keep track of how many disks they have and where their data is.

How does the system manager keep track of disks and make sure they are backed up.

Buy three users 1 TB disk and they may only use 100Gbytes now and will not reach 300GBytes for six or seven months. Buying too early and you are wasting money.

Recycle to smaller users?

A serious management problem

Solution introduce a unified disk system.

Raid

Linux mount points

A linux system has a “root” at /
On this system a number of directories can hang
/home /usr /var /local

They can be directories on a disk,
but any directory can also be a mount for another disk.
So /home can be a completely separate disk.
/home/kyberd might be a directory on this “home” disk
while /home/ellis may be a mount point for a completely separate disk.

Linux allows a single directory structure

This starts to create a single uniform directory space. It is not completely transparent.
If I have 200Gb on /home/kyberd and I want an increase to 500Gb
I can add a bigger disk and copy files across.
Make a mount point /home/kyberd/data

Management issue

System mount points /mnt/disk1 … /mnt/disk2
Soft links from /home/kyberd/teaching -> disk1 etc.

Apparently have disk space but unable to add files in some directories

Make a number of disks appear as a single disk with a single size.

Raid

10 RAID

Redundant **A**rrays of **I**nexpensive **D**isks

Management issue

Large disks were expensive smaller disks were cheap. RAID was originally about cost.

Cheap disks are more likely to fail; more disks and failure more likely.
Expensive disk with an annual failure rate of 0.1%.
Cheap disks with an annual failure rate of 0.2%. But if any of the cheap disks fail the system fails and the joint failure rate is nearly 1.2%.

Expensive disks have a higher areal density (large capacity) and higher spin speed.
Lower latency and faster transfer.

BUT

If we do or three simultaneous reads, and if the reads come off different disks, the cheap option might be faster.

If we replace 1 disk with 0.1% failure with 2 with 0.2% failure but make the two disks identical the failure rate is 0.0004%. (Higher power requirements & more space).

Two low spec disks can give better performance than 1 high spec (for certain applications).
May not be cheaper

The disk controller can choose to read off different disks

Where do you want performance?

Interesting case study: performance is not just a hardware issue. It is how the raw speed is utilised.

RAID 0 :Just a set of disks. No redundancy, no space overheads. May be striped for faster access.

RAID 1: also called mirroring. 1 check disk for every disk. Low space efficiency 50%. No calculation overheads. May be optimised for disk reads, but this may cause writes to take longer.

Reads from either disk, writes to both.

Mirroring precedes RAID

RAID 2: Based on error correcting memory. 1 check disk per 2 data disks. Data split at the bit level. Not used, because of heavy overheads and more use of onboard error correction.

Recovery from single disk failure.

RAID 3: Bit interleaved parity. Each bit from a byte is written to a separate disk. One extra disk for eight data disks takes the parity bit for each byte. Can recover from any single disk failure. Very little overhead for the check. Good performance for read/writing large files. Not so good for small /random I/O.

Recovery from single disk failure.

Raid

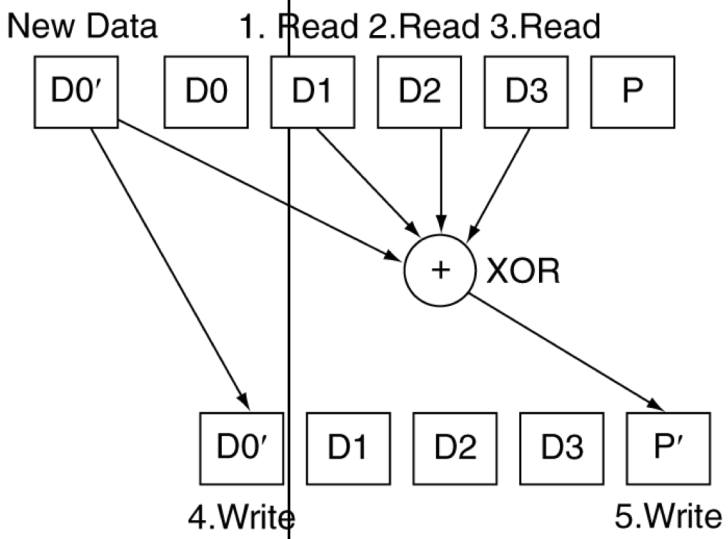
Where do you want performance?

Recovery from single disk failure.

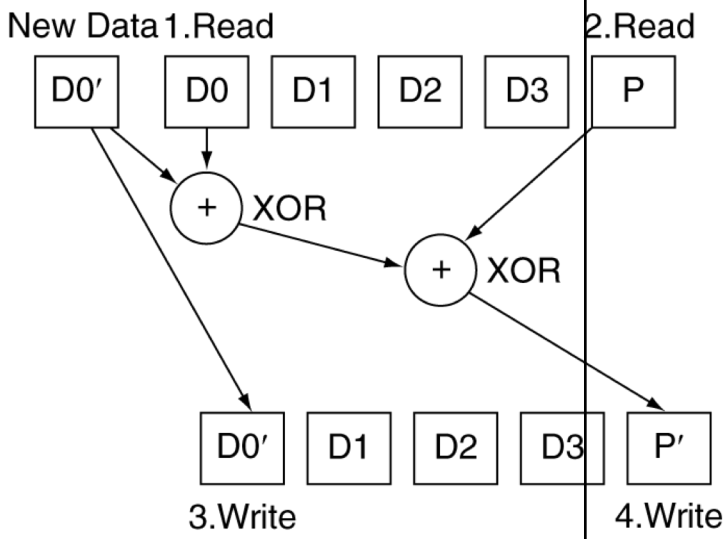
RAID 4: Block-interleaved parity. An enhancement to make RAID 3 better for small accesses. Again uses a single parity disk (which can constitute a pinch point, but it writes blocks on disks not bits (relying on onboard checking. Means that disk reads are independent. Parity disk can be a bottleneck. Disk writes take longer on naïve RAID 4, because writing a block means reading the corresponding block on the other three disks and calculating the parity block and then writing that. 4 reads and 2 writes. Improvement. Read existing data and existing parity. Calculate differences between old and new and from that update the parity block. 2 reads and 2 writes.

Improvements complicate electronics increase cost.

RAID3.



RAID4.



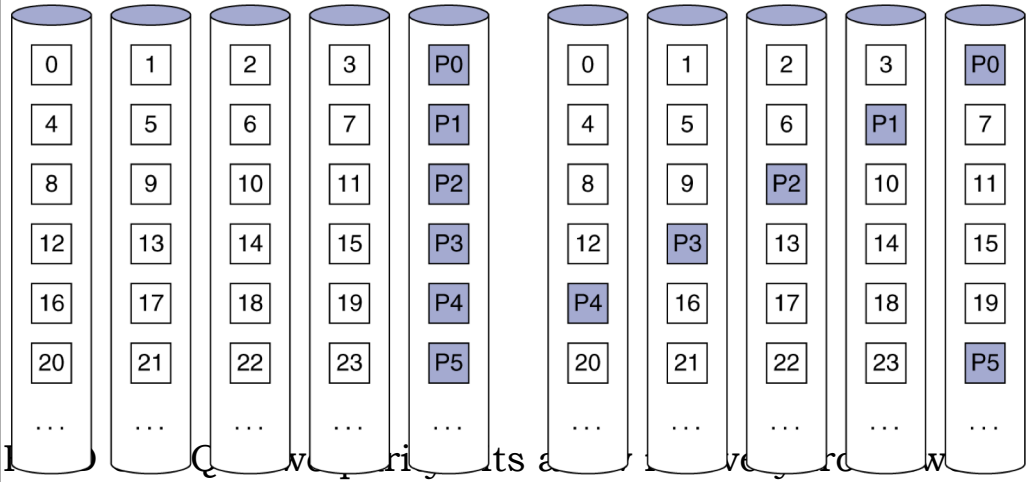
13 RAID Levels

Where do you want performance?

Recovery from single disk failure.

RAID 5: RAID4, but with the check block spread evenly across all five disks. Removes parity disk bottleneck. Benefits of both 3 and 4, but complicated controller. Widely used

If you are dominated by large read/writes RAID3 may still be best



failures

RAID 01 v 10: With 2n disks, do you RAID0 n of them and mirror on the other n RAID 01: “mirrored stripes”. Mirror them in n pairs and stripe across the n. RAID 10: “striped mirrors”

Recovery from single disk failure.

Or RAID 0+1

RAID 1+0

Raid

Multiple disk failures

Large disks increase time for raid rebuild

Companies are introducing systems which allow "hot replacement".

Because a single disk on a Raidset may fail silently, must have alert

Remove bad disk, insert new disk. Rebuild disk for the information on other disks in the Raidset, without the disk becoming unavailable.

Clearly this can reduce the MTTR to zero (if we ignore dual failures). Dual failures are more common than the square of the single failure rate.

Correlated failure – manufacturing defect. Environmental problems.

1 TB disks can take days to rebuild – possibility of second failure. Especially since fails are not always independent.

If reliability and performance are crucial. Need to benchmark during a rebuild.

RAID for multi-core

Raid is a number of disks which act as a single disk.

The result is a system which provides better performance and greater data security

It doesn't work if two or more cores which to access a dataset on a single disk, if the cores themselves are spatially separate.

We need a system where a data set is seen as a single dataset, which is located on a number of different disks. Where those disks may be spatially separate. Hadoop is one such solution

The resulting multiple copies, this time at separate locations mean that it may not be necessary to have a separate backup for the data.

True of most distributed file systems

Could choose one of a number of distributed systems, but I will choose one you can easily learn to use.

Raid

Hadoop a framework with two elements

HDFS – Hadoop Distributed File System

Map-Reduce

Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, Apache Storm.

Hadoop is written in Java (with a little bit of C)

Supports:

C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and Ocaml

Widely used:

Yahoo, Bing, Amazon, Adobe, IBM Cloud, Facebook, LinkedIn, Twitter

- Apache Lucene (search engine) – Doug Cutting
- Nutch – web crawler engine – 2002
- Google File System – GFS – 2003.
  good for Nutch
- Nutch Distributed File System (NDFS)
- Google publish Map/Reduce paper 2004
- Map/Reduce working with NDFS (Cutting)
- Projects combined and call Hadoop
  NDFS renamed HDFS
- Cutting moved to Yahoo
- Hadoop an Apache project
- Google move on to Caffiene and Percolator

Caffiene indexing infrastructure

Percolator: Incremental updates to large datasets

Owning, acquiring, analyzing and managing data have suddenly moved from an operational task required by IT to a top corporate priority where information is viewed as a strategic asset. ….

The attraction of the low-cost, high-availability storage and processing power of Hadoop has drawn many organizations to give this new technology consideration

Key drivers for Hadoop adoption include low-cost data storage coupled with a distributed processing environment that's ideal for experimentation with large, unstructured data sets that have not been accessed by organizations in the past.

Not suitable for real time processing – NoSQL is a solution for this.
Use growing fastest for unstructured data, where there is no existing mature competitor

Hadoop File System

A purpose built distributed file system.
To the client looks like a normal file system.

Since it looks like a normal file system it is easy to use.
But it actual structure allows it to provide a substrate for distributed processing.

How to distribute data?

We could create replicas of the whole file and distribute them around a number of data centres

BUT

It would consume a lot of bandwidth to copy all the data and it would take up a lot of space at the data centres.

So Hadoop divides the data up into “blocks”
The blocks are replicated and each block instance is stored in a number of different places.
A job analyses data from a “near by” storage unit

Some modern tanks have a control system based on Playstation handset.

Read large data sets rapidly (even at the expense of slow read)

To read

- Locate block
- Read block

Transfer time = seek time +

(data size/transfer speed)

Typical seek times are 5ms and a fast transfer is 100 MB/sec

Disk block sizes are typical ½ to 4 Kb

1GByte file will take 10s to read the data.

It will require 250,000 seeks for a badly fragmented file. About 20 minutes

Hadoop default block size is 64Mb

So 16 seeks/Gbyte – about 80ms

(Normal text and programme files are a few kb – with such a large block the waste of space would be enormous)

Can choose larger block size.

HDFS stores blocks – normal filesystems store files.

Blocks are simple buckets of data. The metadata to tie them into files is stored elsewhere.

HDFS automatically replicates data onto different disks – typically 3 servers

no requirement for RAID

Hardware raid is expensive and is often the first thing to break

HDFS does not require expensive reliable hardware.

HDFS uses a master/slave architecture

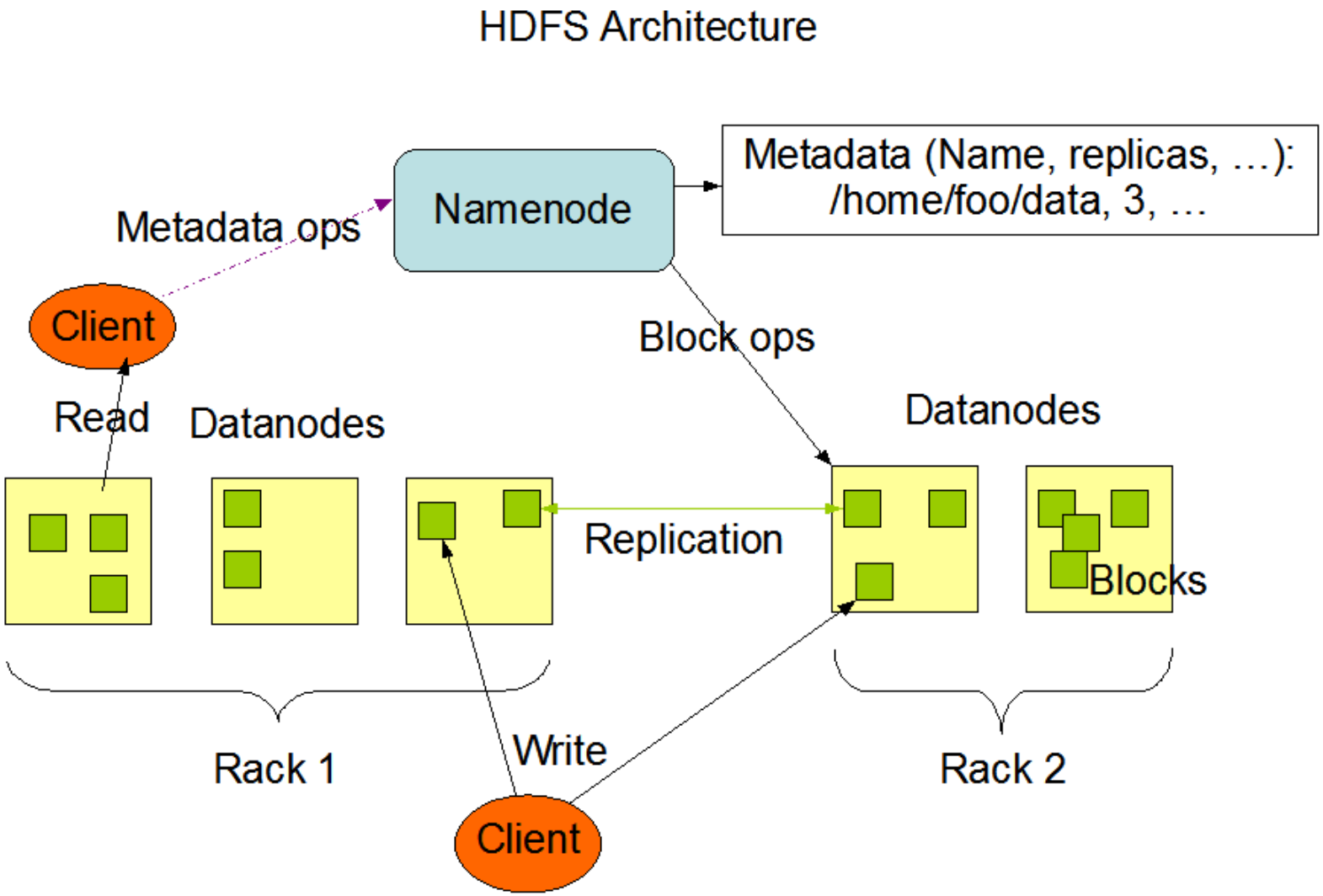
Namenode – single master: manages filesystem space and metadata

Datanodes – many slaves where all the blocks for a given tree are located

Namenode and datanode are programmes that typically run one per machine

Each server in a rack has its own datanode and one runs the namenode

http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html



Each server in a rack has its own Datanode and one runs the Namenode

Client interacts with the Namenode to get references to the data block.

Data is fetched via references to the appropriate Datanodes.

So the load on the Namenode is limited and since the number of Datanodes scales with the amount of data, the response time scales well

The Namenode contains

- the filesystem tree
- metadata for all files and directories
- Reference to the Datanodes to which all the blocks for a given file are stored

Namenodes use two files

- An image of the system
- A transaction (edit) log

On power up the Namenode accesses the Datanodes and rebuilds the block/Datanode map.

It has a filesystem and builds the mapping from the files to the physical locations of the blocks which constitute the file

Datanodes send a heartbeat and a block report (blocks for each file) to the Namenode

Namenode is a single point of failure.

Put all your eggs in one basket and watch that like a hawk

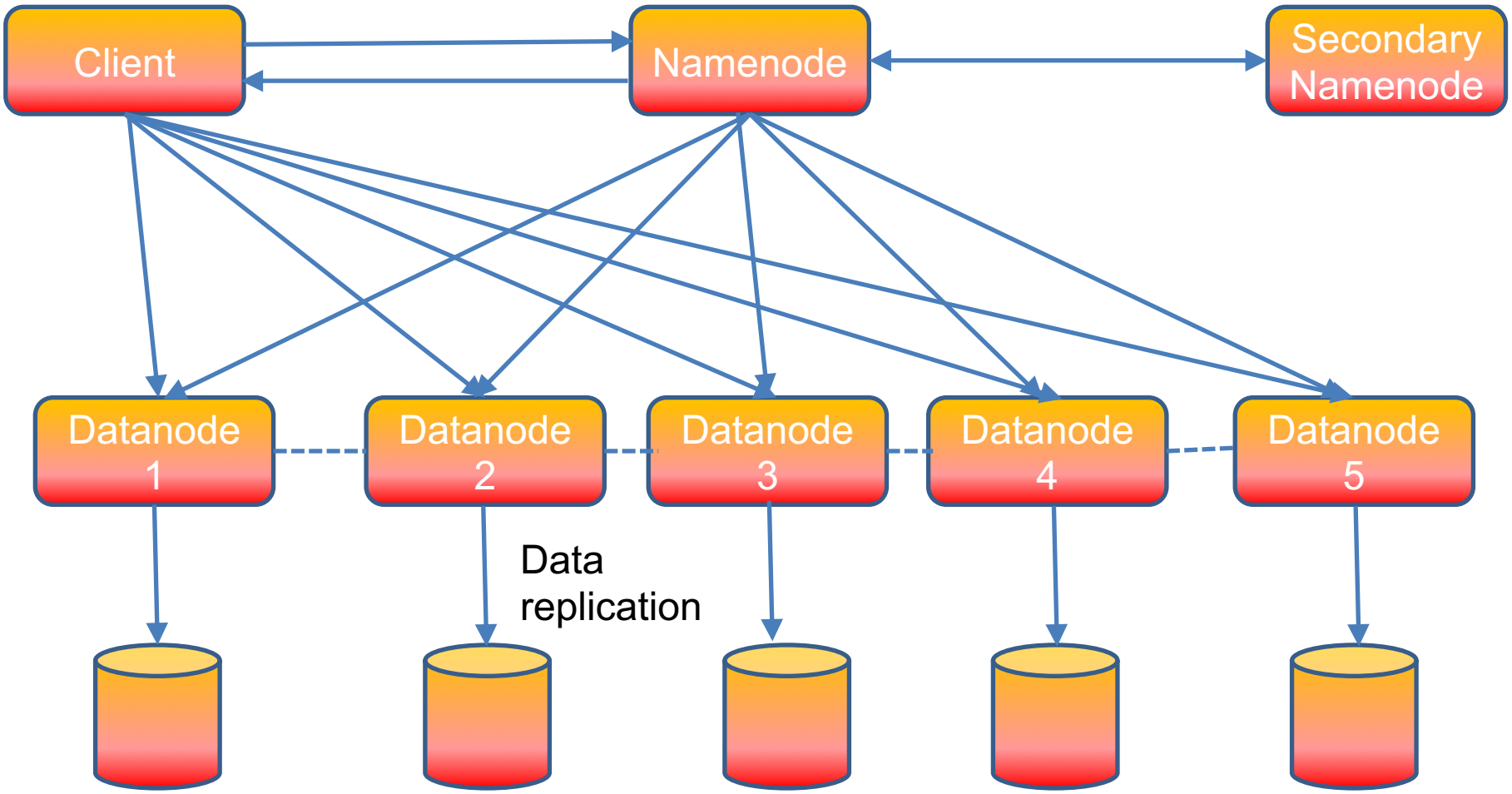
Protect that point of failure

Hadoop provides a means to make a backup of the Namenode files, which constitute a snap-shot state of the system. This is done synchronously to multiple local and remote disks.

A secondary Namenode can be created. Its role is to merge the edit log with the namespace image to prevent the edit log from becoming too large. It also maintains a copy of the merged state. Note: It does not do the same job as a regular Namenode!

Possiblity of some data loss if a Namenode goes down

Namenode is a single point of failure.



"The placement of replicas is crucial to HDFS reliability and performance"

haddop.apache.org

Number of replicas may be specified at creation, and may be subsequently modified.

The default is three replicas.

Uses rack aware placement … in a cluster there is normally higher bandwidth between elements in the same rack, then between racks.

HDFS compromise, one replica in the same rack, one in a different one.

HDFS compromise, one replica in the same rack, one in a different one.

Putting one duplicate in the same rack reduces inter rack traffic, while not significantly reducing the reliability of the system.

If you are interested check out hadoop web site

This is referred to as rack awareness and is transparent to the user.

hadoop tries to satisfy a request with the replica which is “closest”

closest = highest available bandwidth

HDFS Not quite like using a standard filesystem

uses the FS shell

```
hadoop fs -mkdir
hadoop fs -ls
```

Hadoop is not limited to HDFS

Written in java with cross language support

Apache provides a software framework which generates interface code to allow cross-language communication

The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages.

http://thrift.apache.org

Supports URI name space and file names
Uniform Resource Identifier

Looks like a web address

hdfs::/nnAddress:nnPort/path

Designed for very large data sets and in particular requirements to stream it and process it is parallel.

In particular it was built for the Map/Reduce paradigm.

Identical code on distributed machines, is sent chunks of data to work on.

No map process can or needs to see all the data, all they want is a well defined chunk, such as a hadoop data block.

Designed with database searches in mind.

Designed with database searches in mind.

It clearly assumes that there will be only one or two processes accessing a particular chunk at any time.

And that the cores are all in the same rack or the same computer room.

If the data is, for instance, spread over more than one centre (or even country), then one might choose to write both duplicates at a distance site, where they can be accessed by different collections of machines.

Or if there are likely to be multiple simultaneous accesses to a block, again spreading them more widely might be useful

Designed with database searches in mind.

But can be used in more general programming situations.

If you are running the same code on many data segments.

In particle physics we need to analyse collisions between sub-atomic particles.

Each collision is separate. Each one needs to be processed by the same code to extract some interesting information.

At the end the information is summed to get a result.

There may be millions of collisions to analyse and a data set which is 100s of Gigabytes.

Split the data up to reside on a number of servers which are connected to a number of processing units.

The Map phase consists of many copies of the same code running to produce the “answer”.

Those ”answers” are all transferred to the “Reduce” part of the programme, which combines the results from all the Maps, to provide a final answer.

Hadoop large block size - default 64Mb

Block oriented (independent)

Automatic replication

Master (Namenode) – slave (Datanode) architecture.

Client contacts Namenode only for the location of the block.

Data transfer between the Datanode and the client.

Scales well

Good redundancy

Good for parallel processing