Uni.lu HPC School 2018

PS11: Big Data Applications (batch, stream, hybrid)

Uni.lu High Performance Computing (HPC) Team

Dr. S. Varrette

University of Luxembourg (UL), Luxembourg

http://hpc.uni.lu

Latest versions available on Github:

UL HPC tutorials: https://github.com/ULHPC/tutorials

UL HPC School: http://hpc.uni.lu/hpc-school/

PS11 tutorial sources: ulhpc-tutorials.rtfd.io/en/latest/bigdata/

Summary

Main Objectives of this Session

- Brief **review of [Big] Data Management**
  - ↪ **Data transfer** in practice
  - ↪ Sharing Data through VCS/**Git**
- Introduction to **Big Data analytics**
  - ↪ Distributed File Systems (DFS)
  - ↪ **Mapreduce**
  - ↪ **Hadoop** and **HDFS**
    - ✓ local (Standalone) Mode
    - ✓ [pseudo-]distributed mode
- In-memory [streaming] analysis with **Spark**
  - ↪ Interactive runs
    - ✓ PySpark, the Spark Python API
    - ✓ Scala Spark Shell
    - ✓ R Spark Shell
  - ↪ Standalone cluster runs

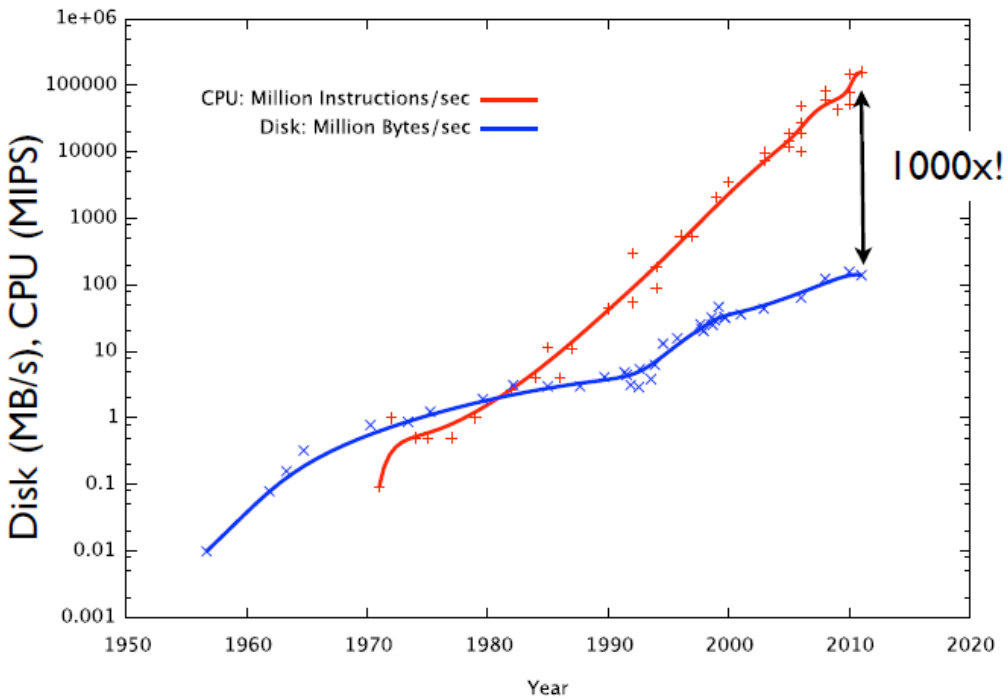
Disclaimer: Acknowledgements

- Part of these slides were **courtesy** borrowed w. permission from:
  - ↪ Prof. Martin Theobald *(Big Data and Data Science Research Group)*, UL
- Part of the slides material adapted from:
  - ↪ Advanced Analytics with Spark, O Reilly
  - ↪ Data Analytics with HPC courses
    - ✓ © CC AttributionNonCommercial-ShareAlike 4.0
- similar hands-on material on Github for instance:
  - ↪ Jonathan Dursi: `hadoop-for-hpcers-tutorial`

Summary

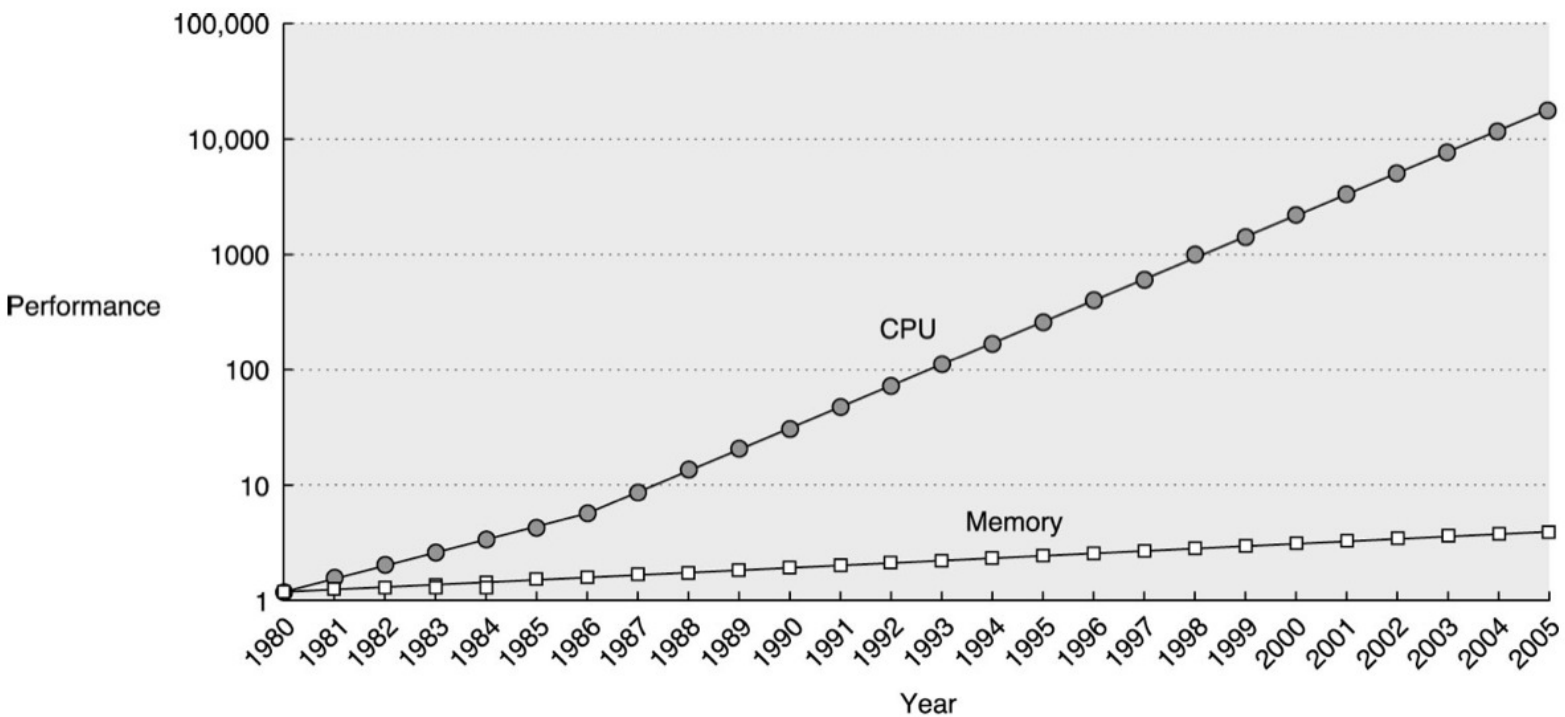
Data Intensive Computing

- Data volumes increasing massively
  - ↪ Clusters, storage capacity increasing massively
- Disk speeds are not keeping pace.
- Seek speeds even worse than read/write



Data Intensive Computing

- Data volumes increasing massively
  - ↪ Clusters, storage capacity increasing massively
- Disk speeds are not keeping pace.
- Seek speeds even worse than read/write



Speed Expectation on Data Transfer

http://fasterdata.es.net/

- How long to transfer **1 TB** of data across various speed networks?

Network	Time
10 Mbps	300 hrs (12.5 days)
100 Mbps	30 hrs
1 Gbps	3 hrs
10 Gbps	20 minutes

- **(Again)** small I/Os really **kill** performances
  - ↪ **Ex**: transferring 80 TB for the backup of `ecosystem_biology`
  - ↪ same rack, 10Gb/s. 4 weeks ⟶ 63TB transfer...

Speed Expectation on Data Transfer

http://fasterdata.es.net/

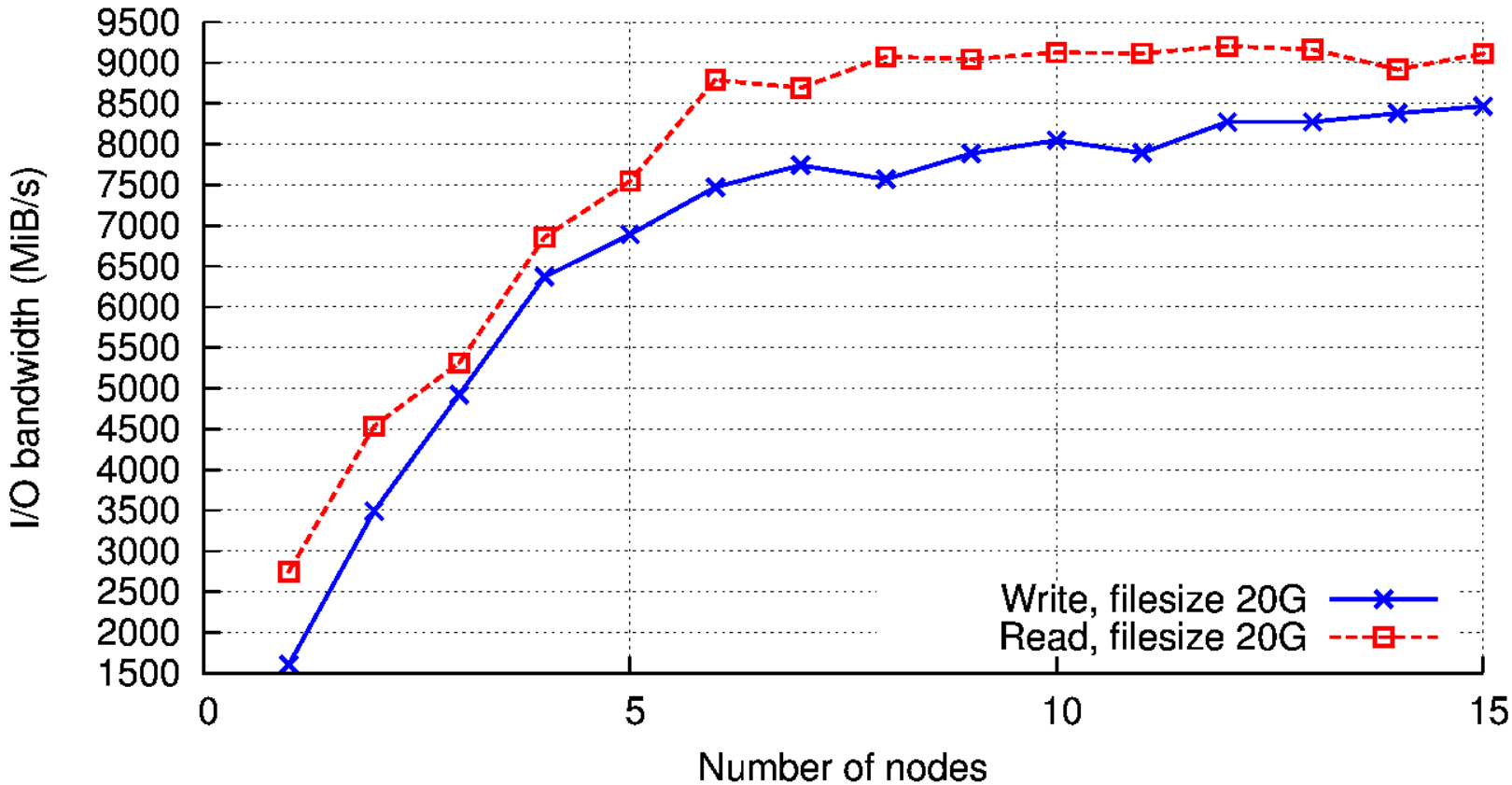
Data set size	1 Minute	5 Minutes	20 Minutes	1 Hour
	Time to transfer			
10PB	166.67 TB/sec	33.33 TB/sec	8.33 TB/sec	2.78 TB/sec
1PB	16.67 TB/sec	3.33 TB/sec	833.33 GB/sec	277.78 GB/sec
100TB	1.67 TB/sec	333.33 GB/sec	83.33 GB/sec	27.78 GB/sec
10TB	166.67 GB/sec	33.33 GB/sec	8.33 GB/sec	2.78 GB/sec
1TB	16.67 GB/sec	3.33 GB/sec	833.33 MB/sec	277.78 MB/sec
100GB	1.67 GB/sec	333.33 MB/sec	83.33 MB/sec	27.78 MB/sec
10GB	166.67 MB/sec	33.33 MB/sec	8.33 MB/sec	2.78 MB/sec
1GB	16.67 MB/sec	3.33 MB/sec	0.83 MB/sec	0.28 MB/sec
100MB	1.67 MB/sec	0.33 MB/sec	0.08 MB/sec	0.03 MB/sec

Speed Expectation on Data Transfer

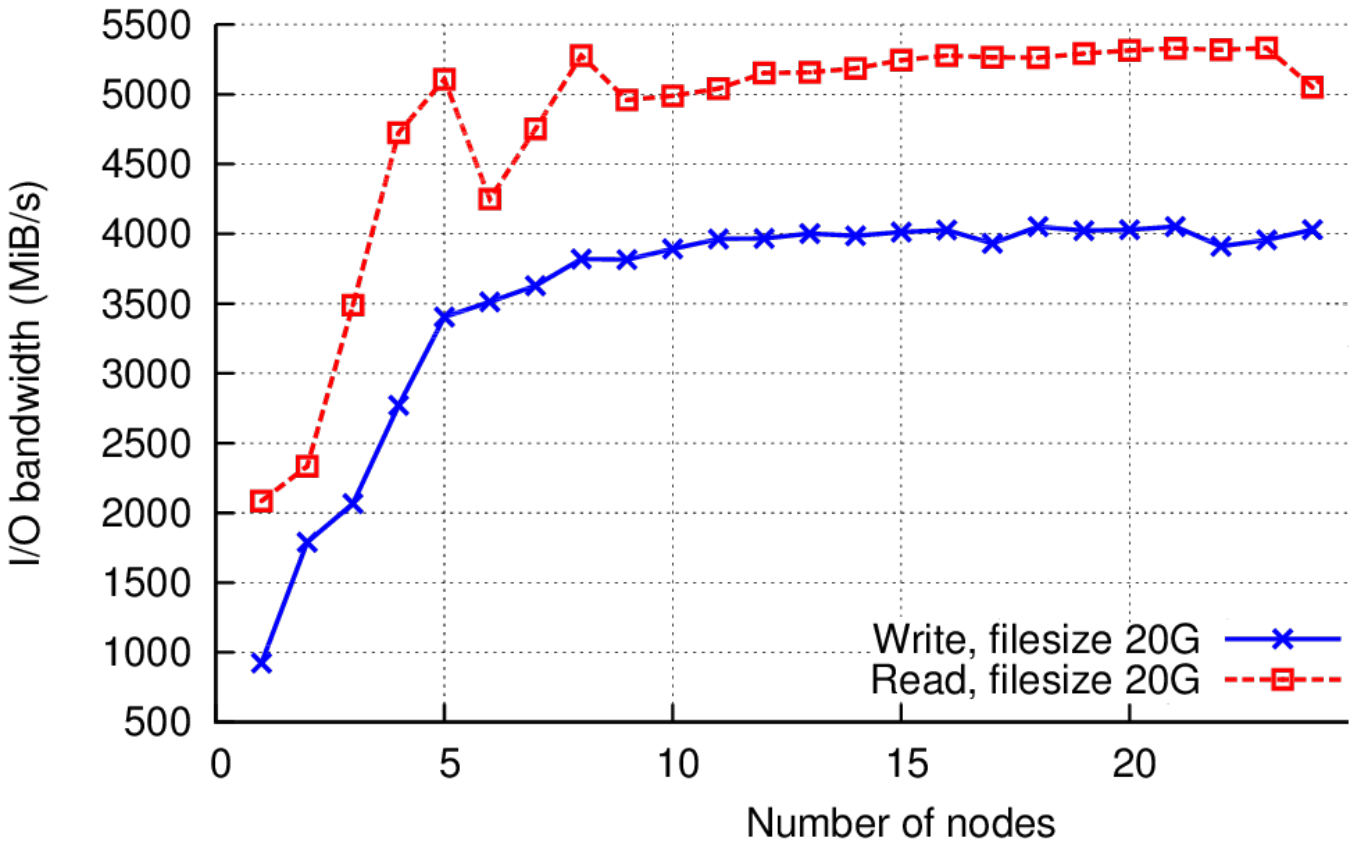
http://fasterdata.es.net/

Data set size	8 Hours	24 Hours	7 Days	30 Days
	Time to transfer			
1XB	34.72 TB/sec	11.57 TB/sec	1.65 TB/sec	385.80 GB/sec
100PB	3.47 TB/sec	1.16 TB/sec	165.34 GB/sec	38.58 GB/sec
10PB	347.22 GB/sec	115.74 GB/sec	16.53 GB/sec	3.86 GB/sec
1PB	34.72 GB/sec	11.57 GB/sec	1.65 GB/sec	385.80 MB/sec
100TB	3.47 GB/sec	1.16 GB/sec	165.34 MB/sec	38.58 MB/sec
10TB	347.22 MB/sec	115.74 MB/sec	16.53 MB/sec	3.86 MB/sec
1TB	34.72 MB/sec	11.57 MB/sec	1.65 MB/sec	0.39 MB/sec
100GB	3.47 MB/sec	1.16 MB/sec	0.17 MB/sec	0.04 MB/sec
10GB	0.35 MB/sec	0.12 MB/sec	0.02 MB/sec	0.00 MB/sec

Storage Performances: GPFS

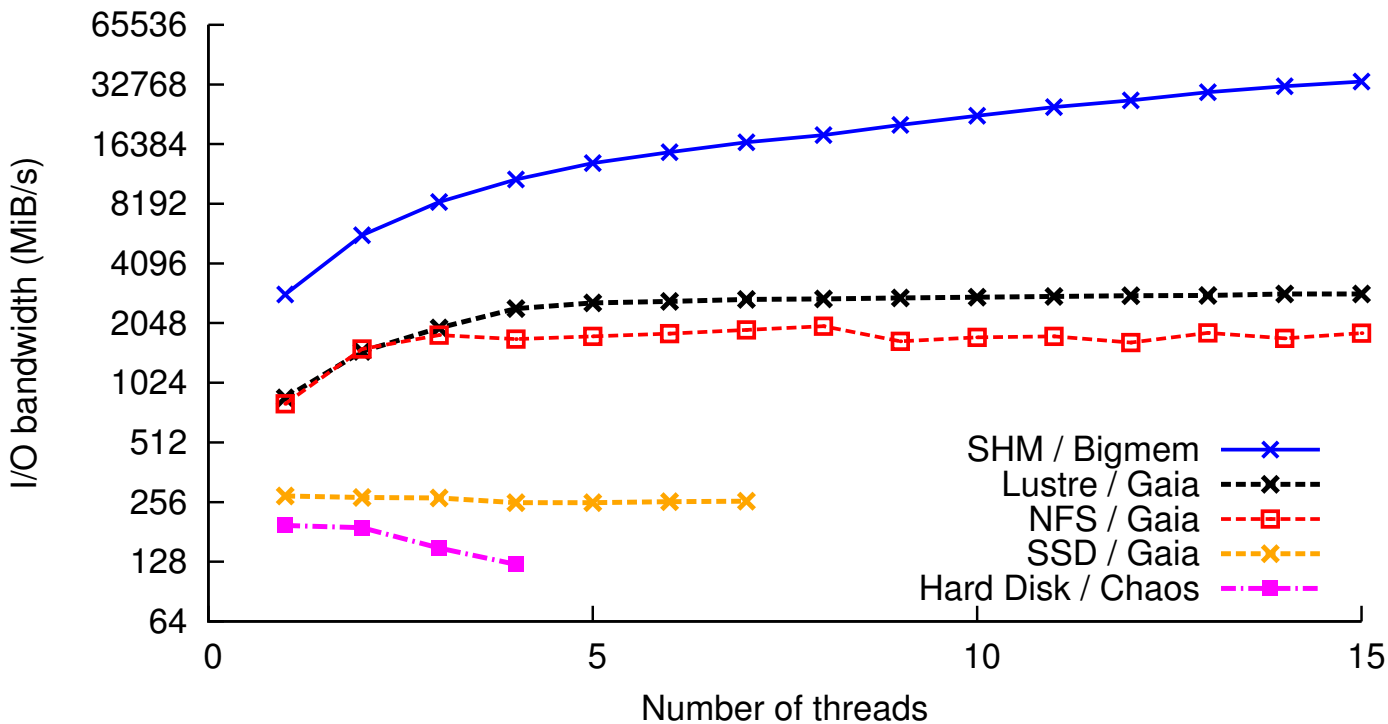


Storage Performances: Lustre



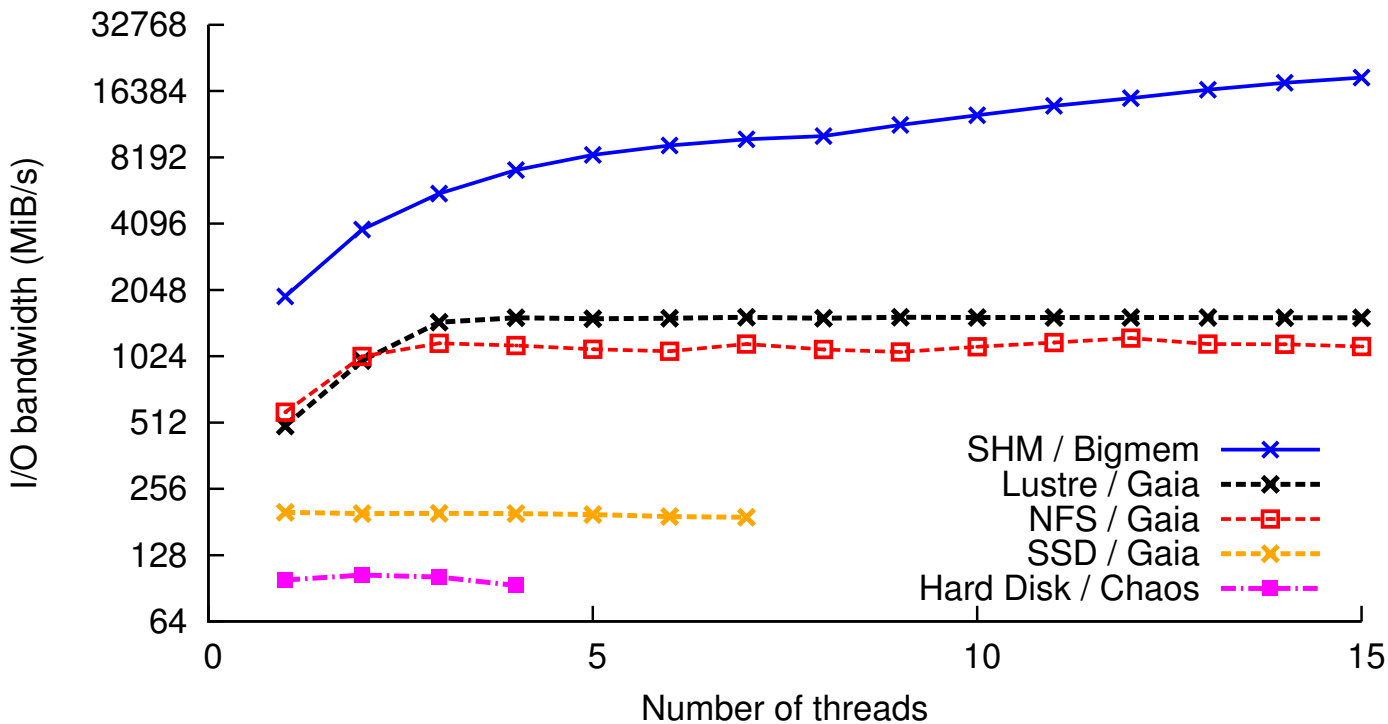
Storage Performances

- Based on IOR or IOZone, reference I/O benchmarks **Read**
  - ↪ *tests performed in 2013*



Storage Performances

- Based on IOR or IOZone, reference I/O benchmarks **Write**
  - ↪ *tests performed in 2013*



Understanding Your Storage Options

Where can I store and manipulate my data?

- **Shared** storage
  - ↪ `NFS` - **not scalable** $\sim\simeq$ 1.5 GB/s (R) — $\mathcal{O}$(100 TB)
  - ↪ `GPFS` - **scalable** $\sim\sim\simeq$ 10 GB/s (R) — $\mathcal{O}$(1 PB)
  - ↪ `Lustre` - **scalable** $\sim\sim\simeq$ 5 GB/s (R) — $\mathcal{O}$(0.5 PB)
- **Local** storage
  - ↪ local file system (`/tmp`) — $\mathcal{O}$(200 GB)
    - ✓ over HDD $\simeq$ 100 MB/s, over SDD $\simeq$ 400 MB/s
  - ↪ RAM (`/dev/shm`) $\simeq$ 30 GB/s (R) — $\mathcal{O}$(20 GB)
- **Distributed** storage
  - ↪ HDFS, Ceph, GlusterFS - **scalable** $\sim\sim\simeq$ 1 GB/s

$\Rightarrow$ **In all cases:** small I/Os really **kill** storage performances

Data Transfer in Practice

```
$> wget [-O <output>] <url>          # download file from <url>
```

```
$> curl [-o <output>] <url>          # download file from <url>
```

- Transfer **from** FTP/HTTP[S] `wget` or (better) `curl`
  - ↪ can also serve to send HTTP POST requests
  - ↪ support HTTP cookies (useful for JDK download)

Data Transfer in Practice

```
$> scp [-P <port>] <src> <user>@<host>:<path>
```

```
$> rsync -avzu [-e 'ssh -p <port>'] <src> <user>@<host>:<path>
```

- [Secure] Transfer **from/to** two remote machines over SSH
  - ↪ `scp` or (better) `rsync` (transfer **only** what is required)
- Assumes you have understood and configured appropriately SSH!

Sharing Code and Data

- Before doing **Big** Data, manage and version correctly **normal** data

What kinds of systems are available?

- *Good*: NAS, Cloud — Dropbox, Google Drive, Figshare...
- **Better** - **Version Control systems** (VCS)
  ↪ SVN, Git and Mercurial
- **Best** - **Version Control Systems** on the **Public/Private Cloud**
  ↪ GitHub, Bitbucket, Gitlab

Sharing Code and Data

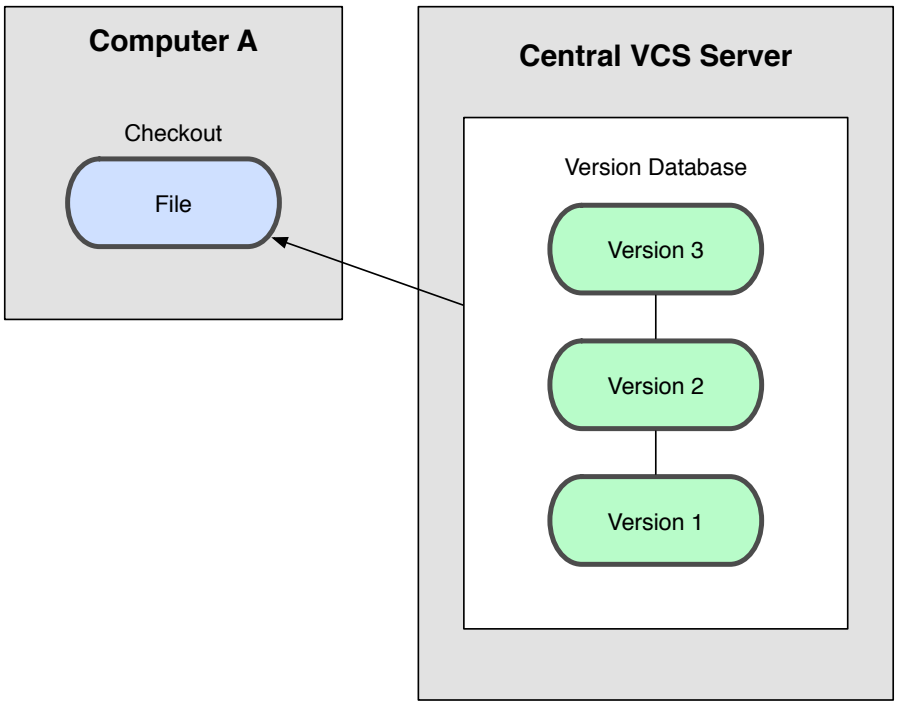
- Before doing **Big** Data, manage and version correctly **normal** data

What kinds of systems are available?

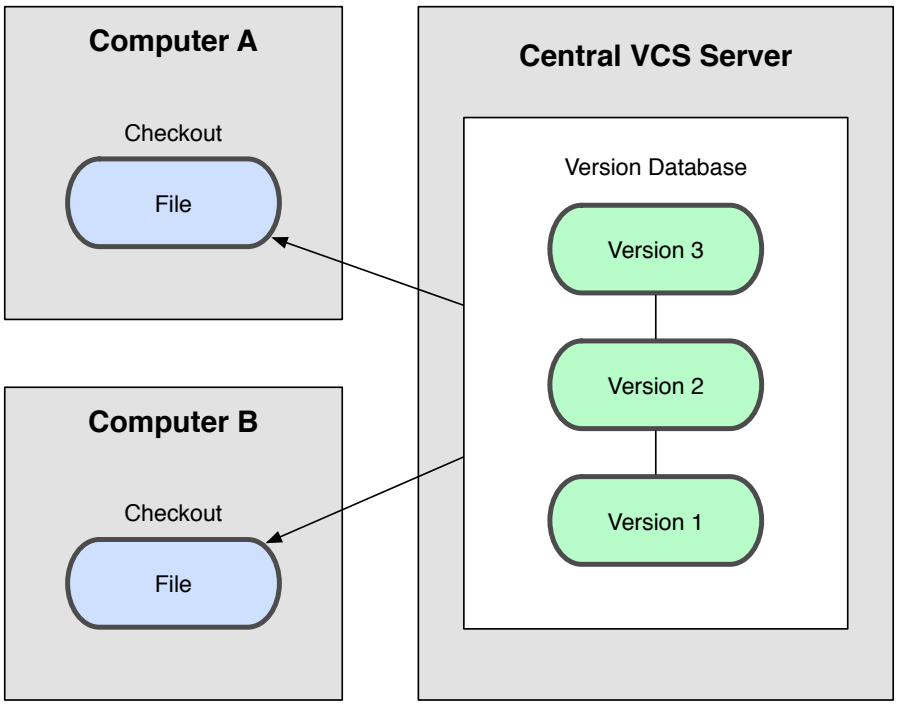
- *Good*: NAS, Cloud — Dropbox, Google Drive, Figshare...
- **Better** - **Version Control systems** (VCS)
  - ↪ SVN, Git and Mercurial
- **Best** - **Version Control Systems** on the **Public/Private Cloud**
  - ↪ GitHub, Bitbucket, Gitlab

- **Which one?**
  - ↪ Depends on the level of privacy you expect
    - ✓ ... but you probably already know these tools ☺
  - ↪ **Few handle GB files...**

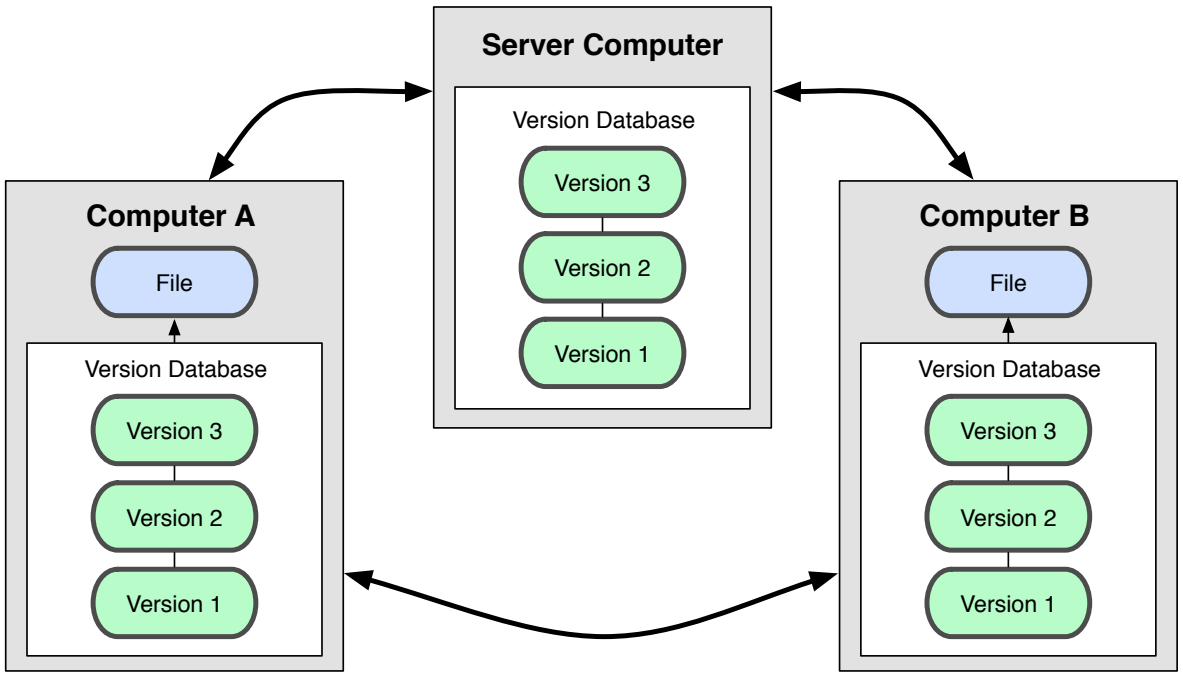
Centralized VCS - CVS, SVN



Centralized VCS - CVS, SVN

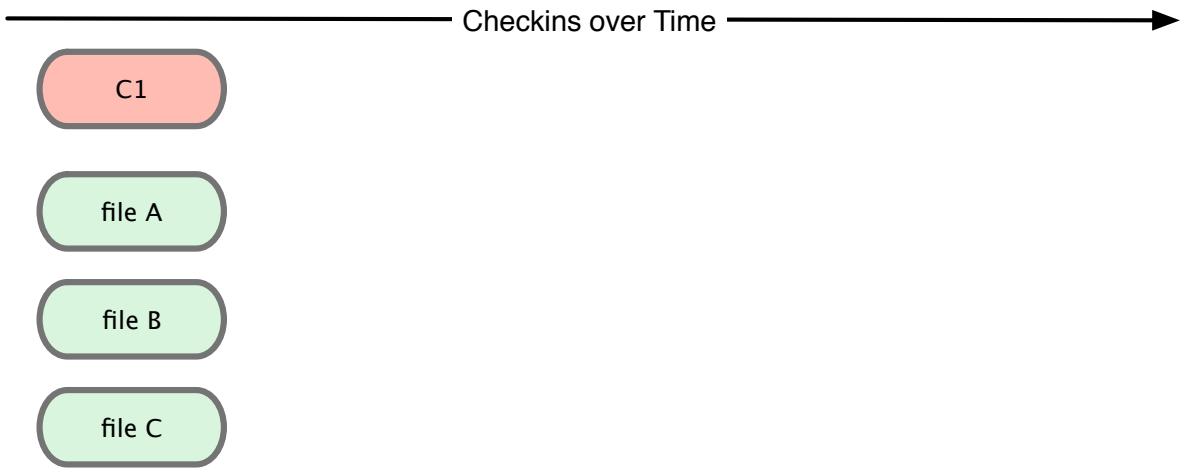


Distributed VCS - Git

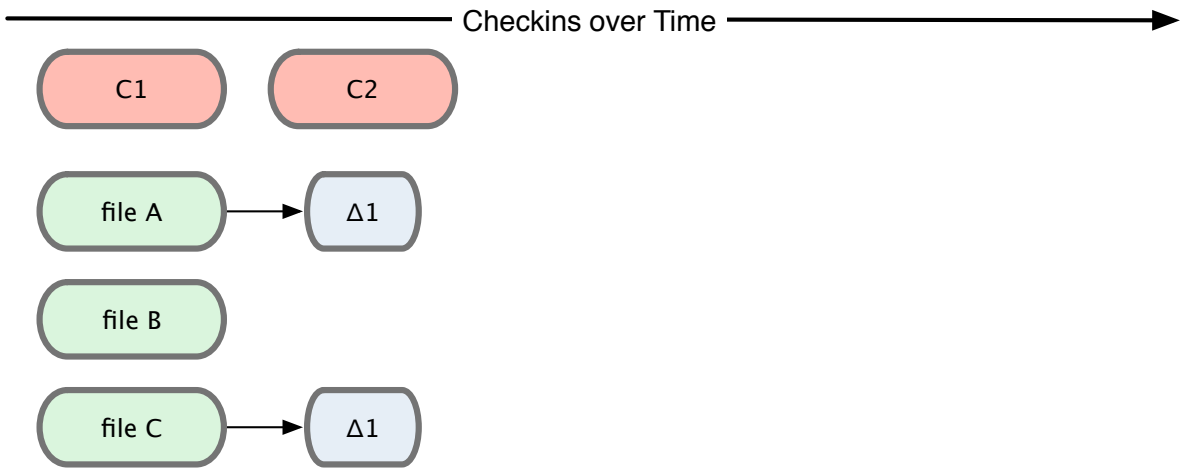


Everybody has the full history of commits

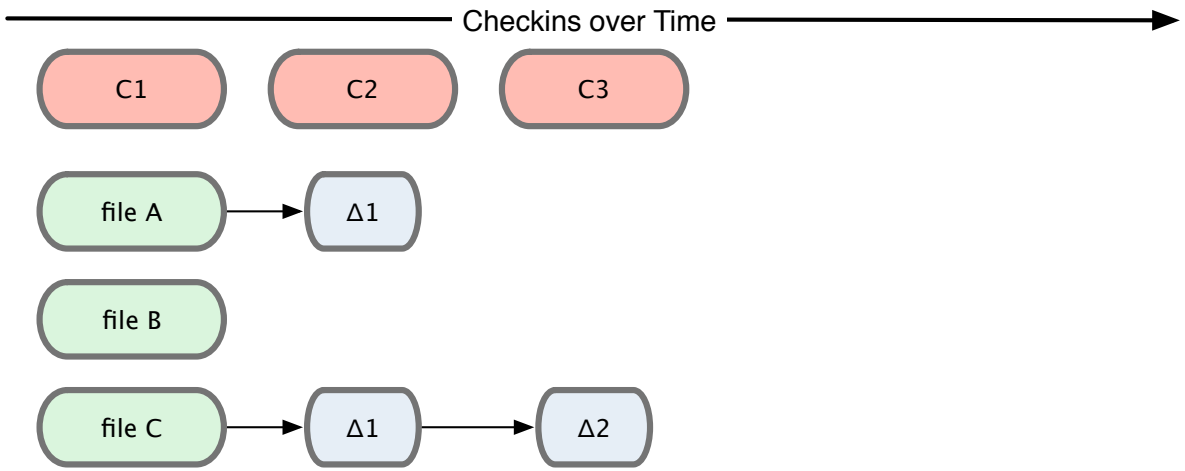
Tracking changes (most VCS)



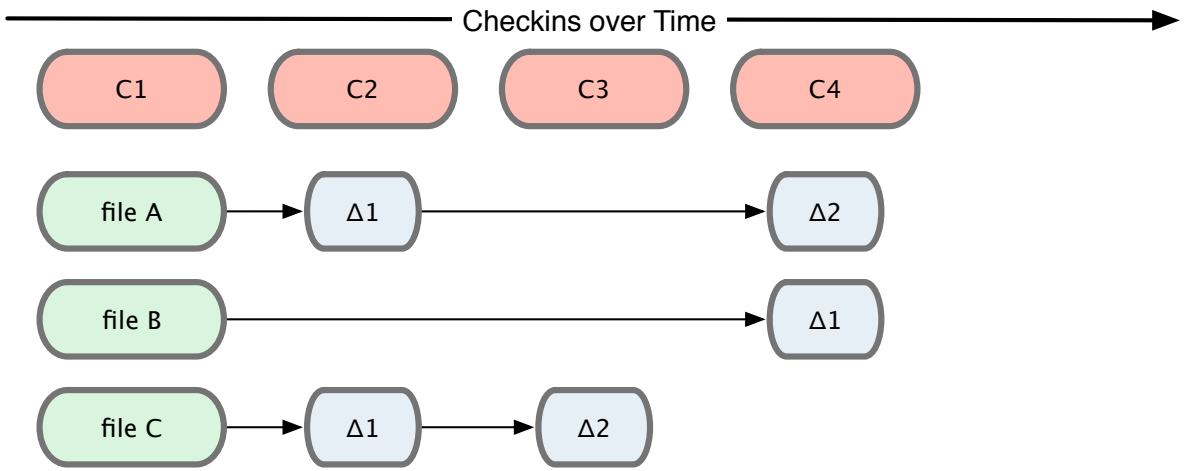
Tracking changes (most VCS)



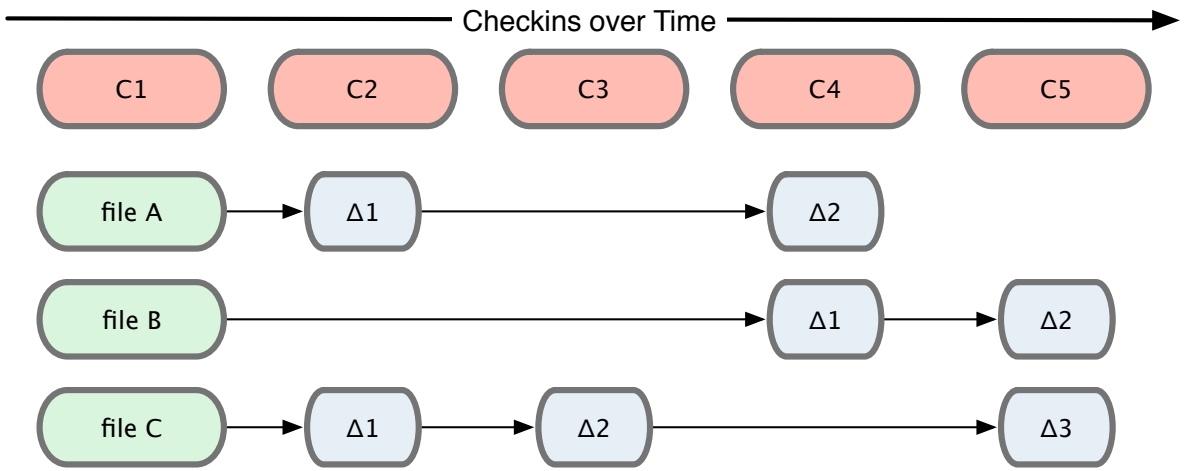
Tracking changes (most VCS)



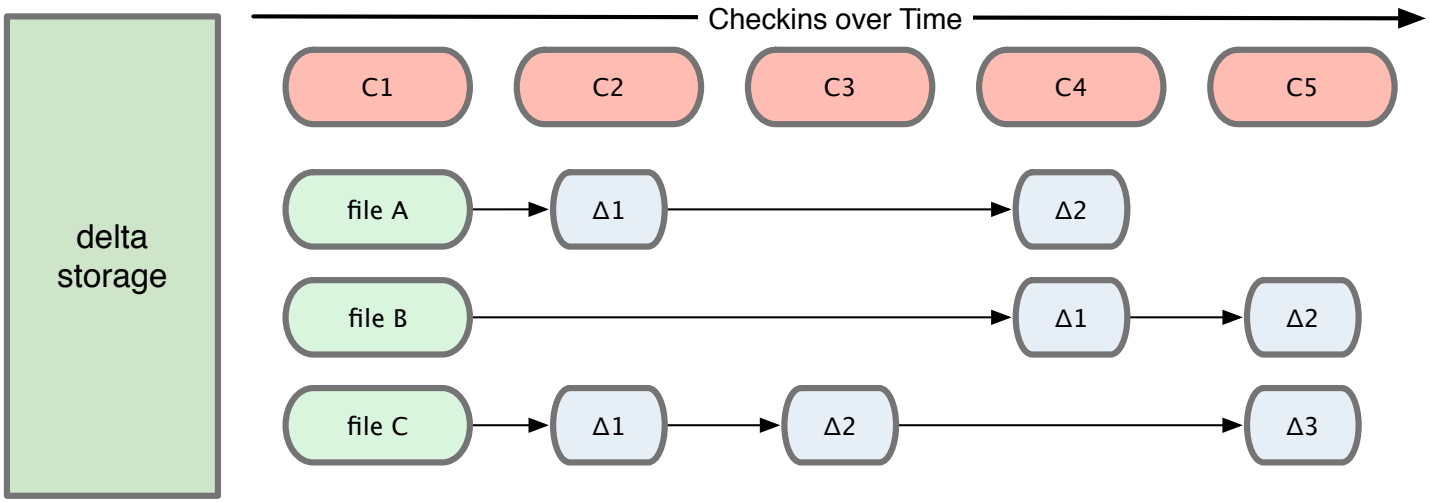
Tracking changes (most VCS)



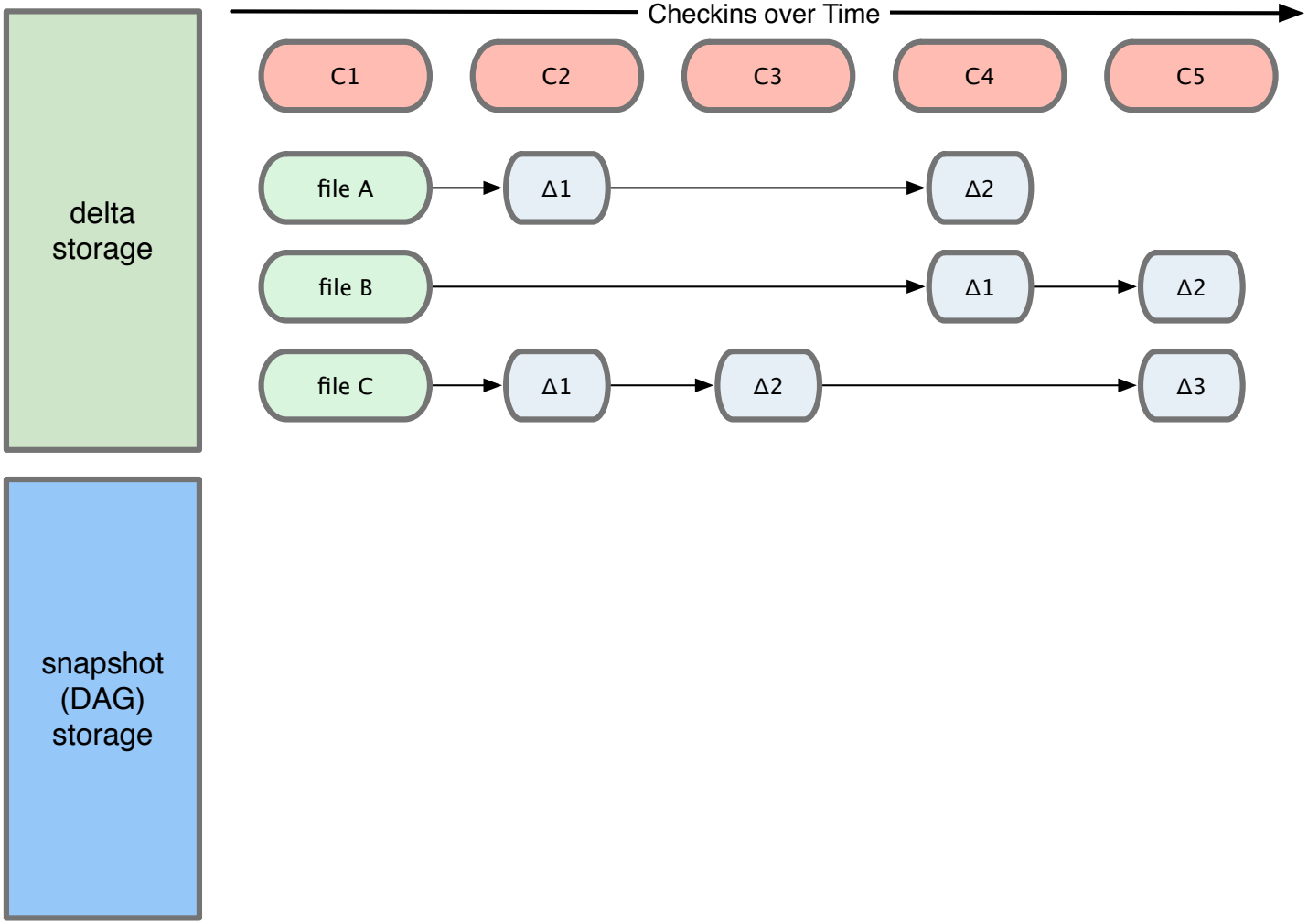
Tracking changes (most VCS)



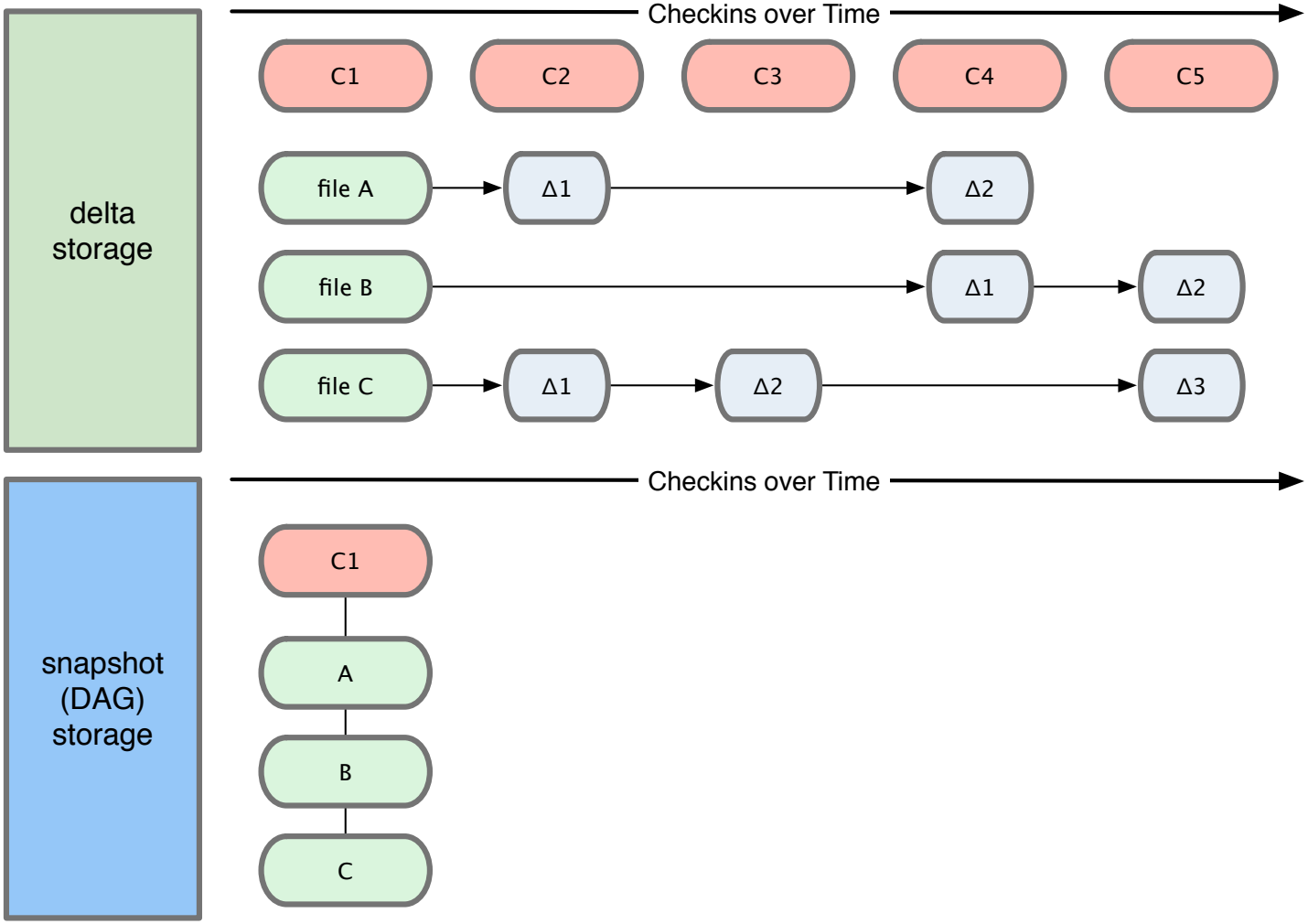
Tracking changes (most VCS)



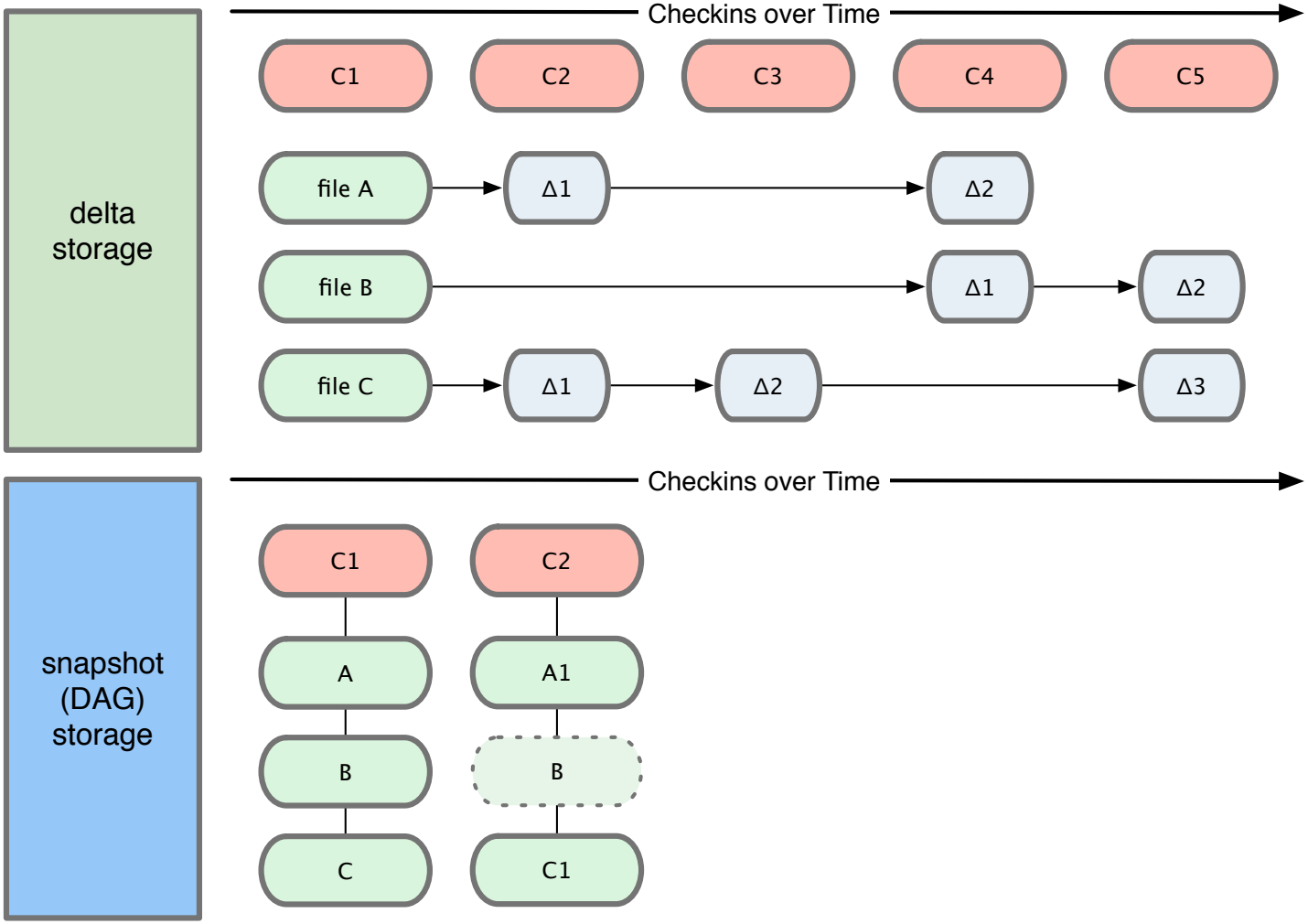
Tracking changes (Git)



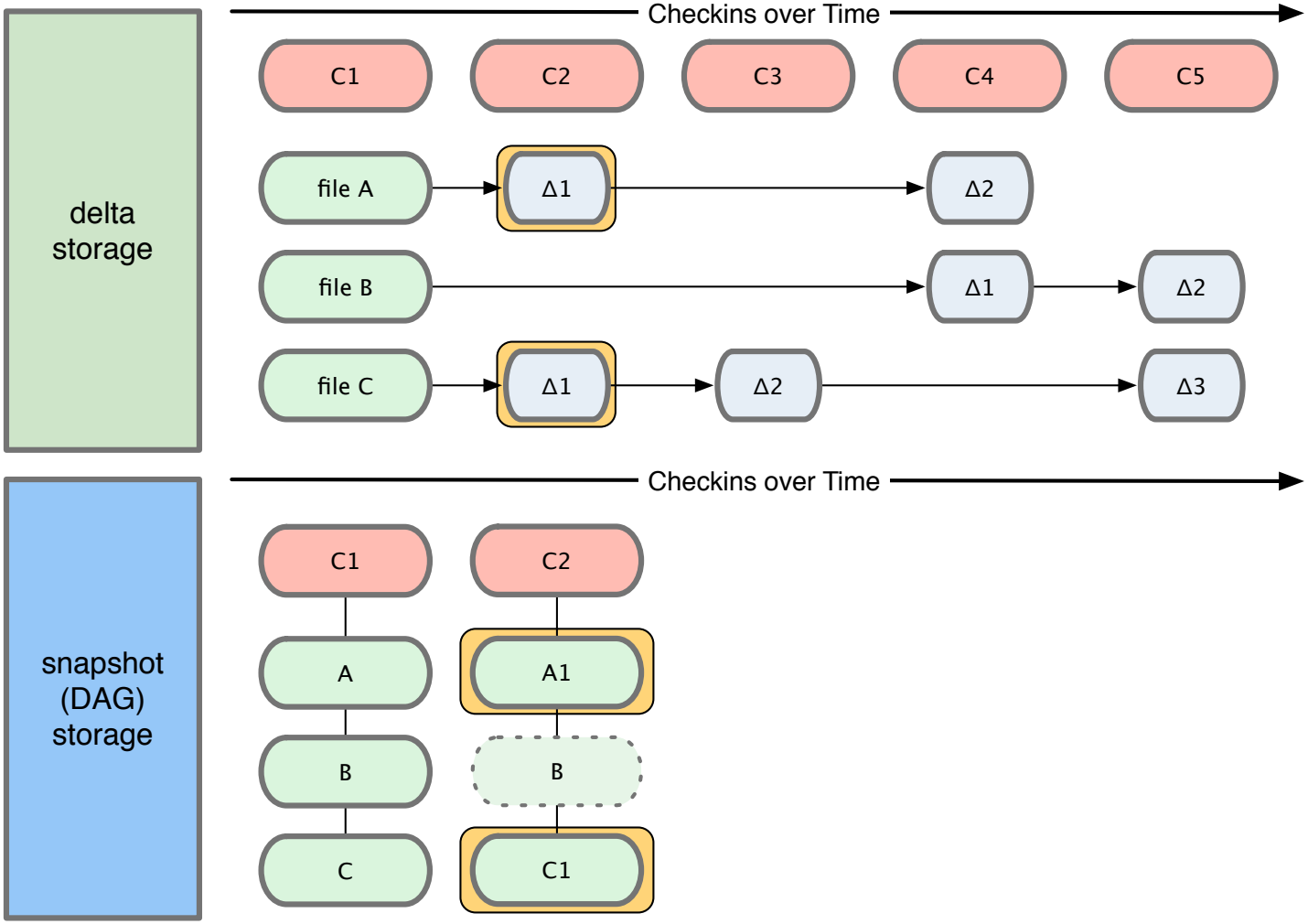
Tracking changes (Git)



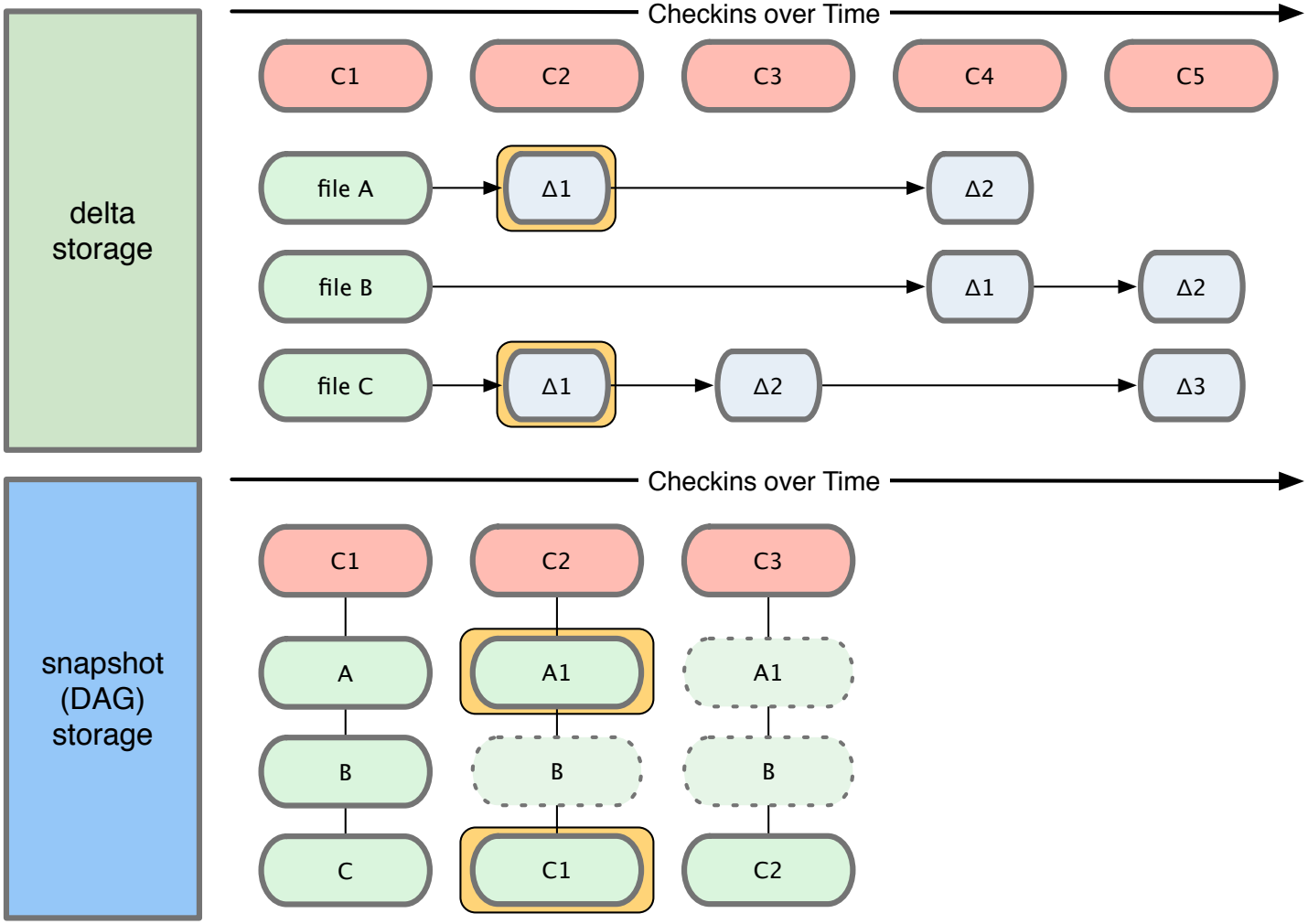
Tracking changes (Git)



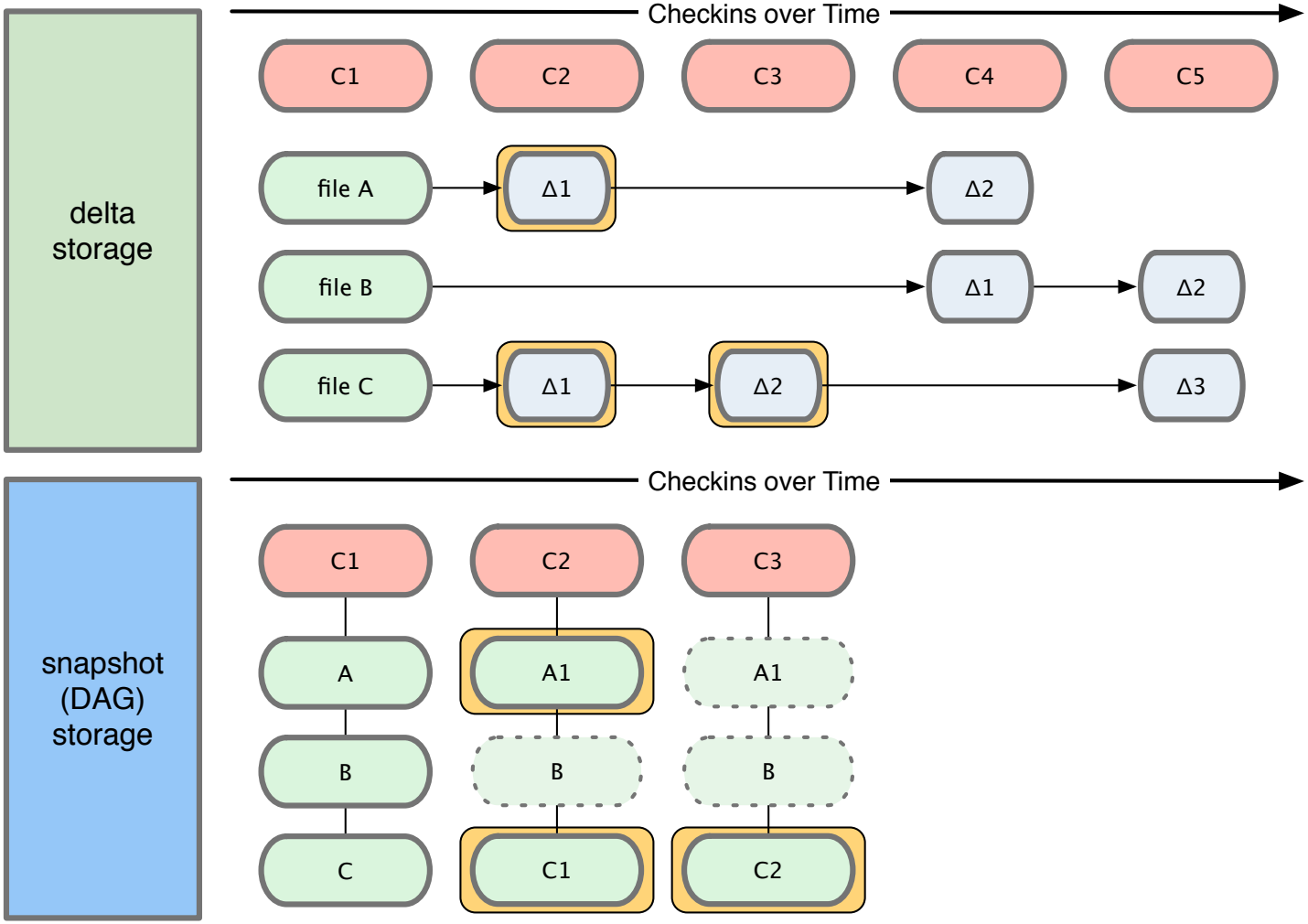
Tracking changes (Git)



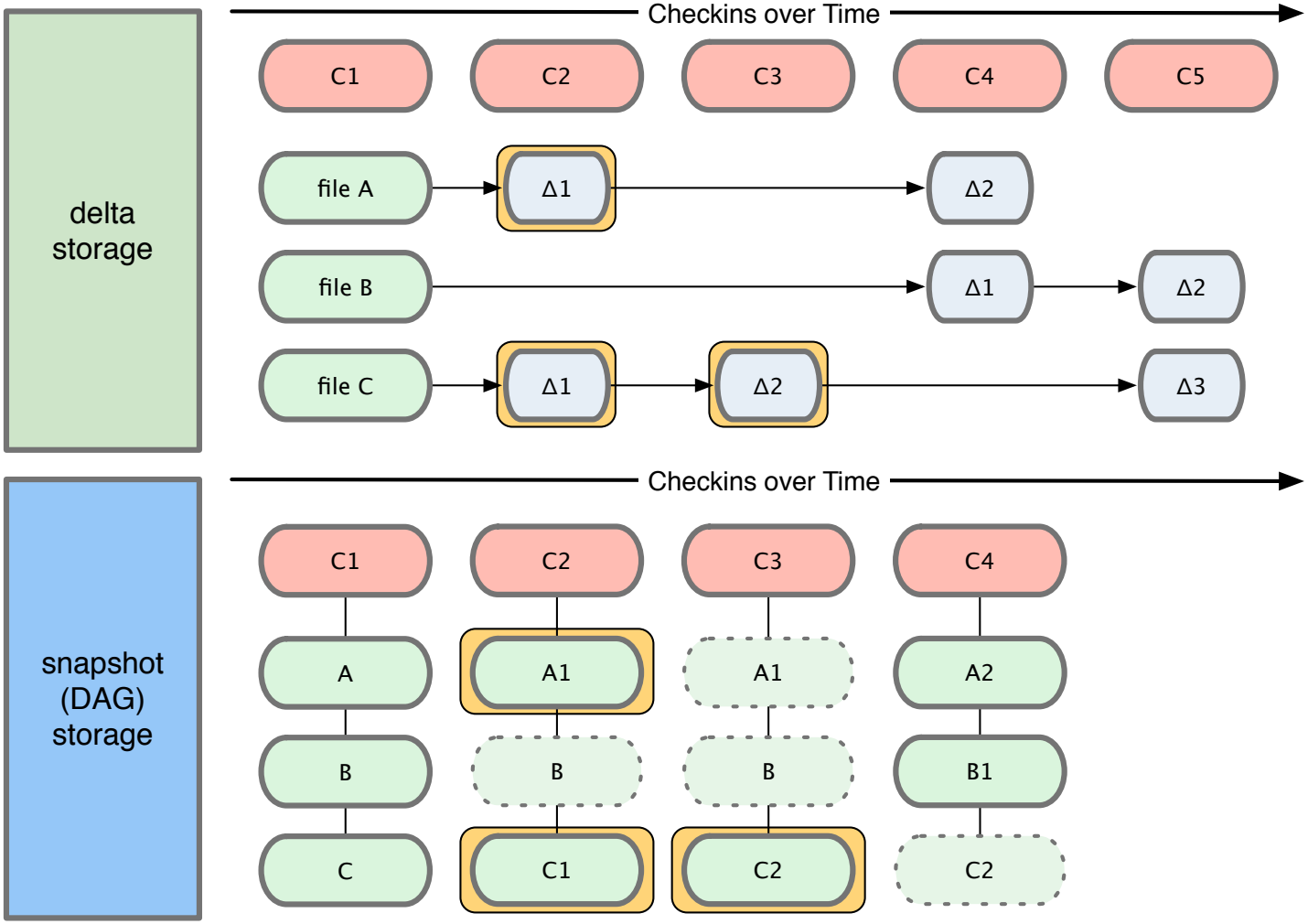
Tracking changes (Git)



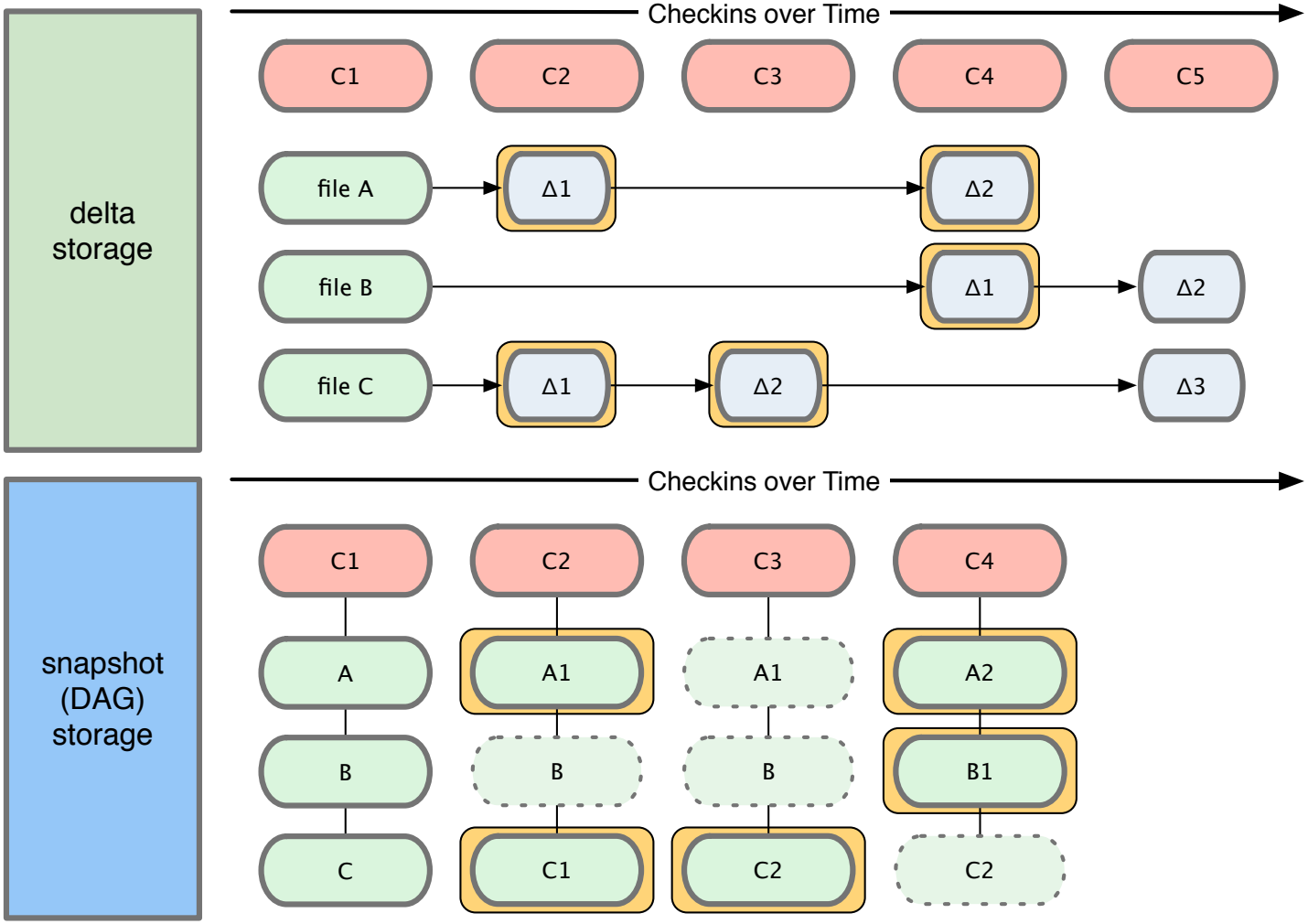
Tracking changes (Git)



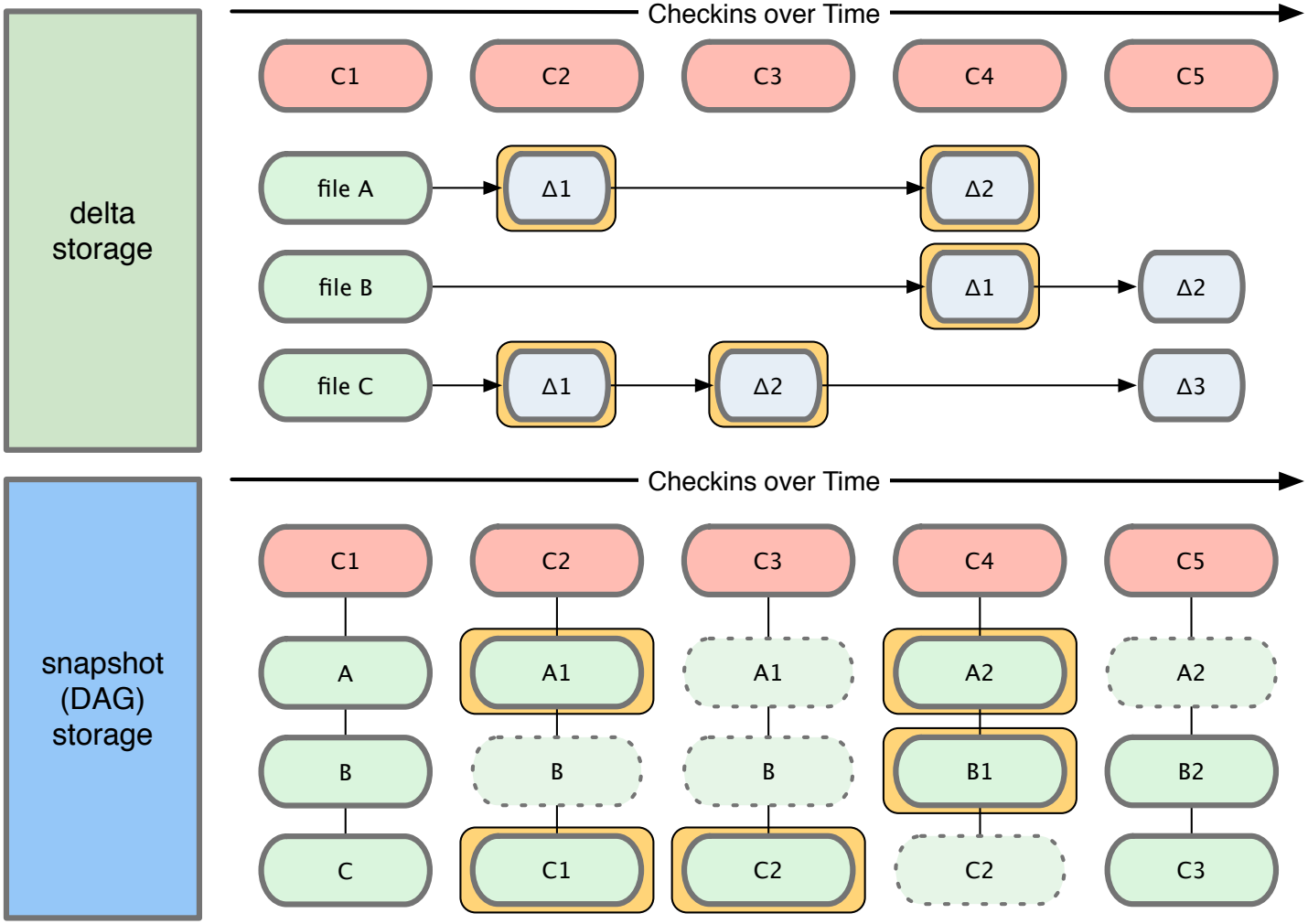
Tracking changes (Git)



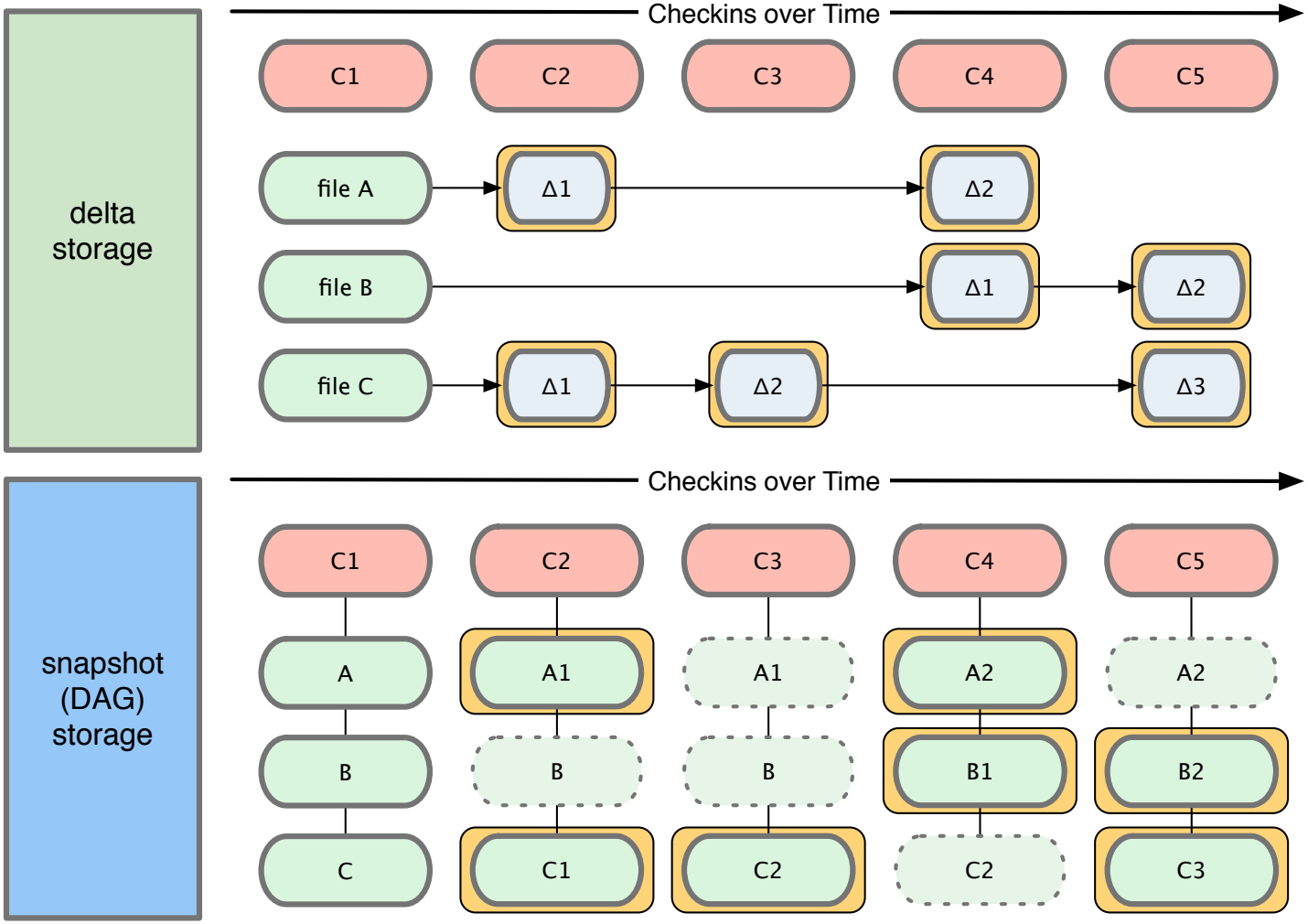
Tracking changes (Git)



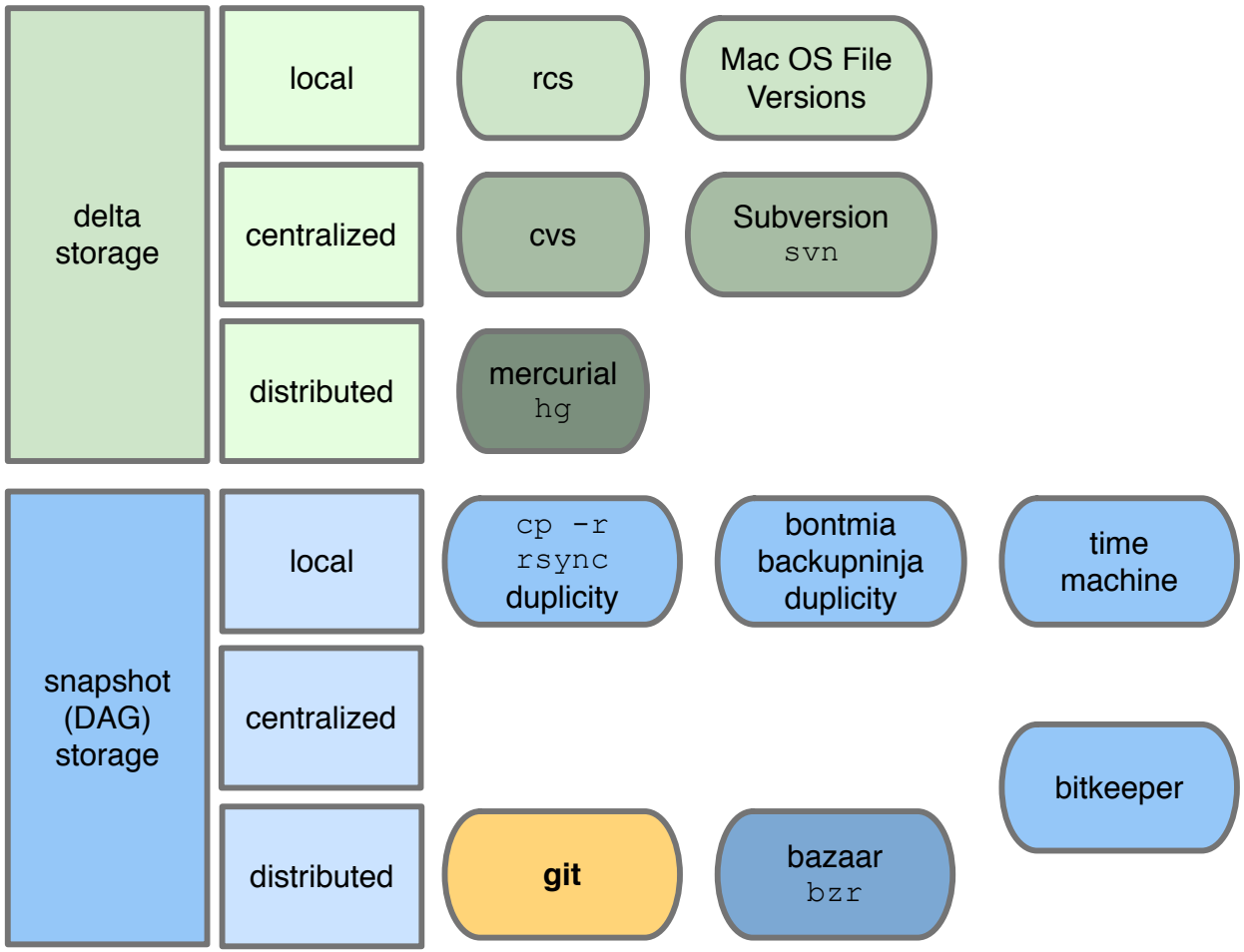
Tracking changes (Git)



Tracking changes (Git)



VCS Taxonomy



Git at the heart of BD

http://git-scm.org

So what makes Git so useful?

(almost) Everything is local

- everything is fast
- every clone is a backup
- you work **mainly offline**

Ultra Fast, Efficient & Robust

- Snapshots, not patches (deltas)
- **Cheap branching and merging**
  - ↪ Strong support for thousands of parallel branches
- Cryptographic integrity everywhere

Other Git features

- **Git does not delete**
  - ↪ **Immutable** objects, Git generally only adds data
  - ↪ If you mess up, you can usually recover your stuff
    - ✓ Recovery can be tricky though

Other Git features

- **Git does not delete**
  - ↪ **Immutable** objects, Git generally only adds data
  - ↪ If you mess up, you can usually recover your stuff
    - ✓ Recovery can be tricky though

Git Tools / Extension

- cf. **Git submodules** or **subtrees**
- **Introducing git-flow**
  - ↪ workflow with a strict branching model
  - ↪ offers the `git` commands to follow the workflow

```
$> git flow init
$> git flow feature { start, publish, finish } <name>
$> git flow release { start, publish, finish } <version>
```

Git in practice

Basic Workflow

- **Pull** latest changes — `git pull`
- **Edit** files — `vim` / `emacs` / `subl` ...
- **Stage** the changes — `git add`
- **Review** your changes — `git status`
- **Commit** the changes — `git commit`

Git in practice

Basic Workflow

- **Pull** latest changes — `git pull`
- **Edit** files — `vim` / `emacs` / `subl` ...
- **Stage** the changes — `git add`
- **Review** your changes — `git status`
- **Commit** the changes — `git commit`

For cheaters: A Basicerer Workflow

- **Pull** latest changes — `git pull`
- **Edit** files — `vim` / `emacs` / `subl` ...
- Stage & commit all the changes — `git commit -a`

Git Summary

- **Advices**: **Commit early, commit often!**
  - ↪ commits = save points
    - ✓ use descriptive commit messages
  - ↪ Do not get out of sync with your collaborators
  - ↪ Commit the sources, not the derived files

- **Not covered here (by lack of time)**
  - ↪ does not mean you should not dig into it!
  - ↪ *Resources*:
    - ✓ `https://git-scm.com/`
    - ✓ tutorial: IT/Dev[op]s Army Knives Tools for the Researcher
    - ✓ tutorial: Reproducible Research at the Cloud Era

Summary

Before we start. . .

- We will rely on EasyBuild to build Hadoop and Spark
  - ↪ assumes you have followed PS5: Easybuild
    - ✓ "Building [custom] software with EasyBuild"
- Building Hadoop is quite **long**
  - ↪ We're going to install the most recent Hadoop by Cloudera
  - ↪ **Thus** we will launch the build in parallel of this talk.
  - ↪ a few **preliminary** software are required

Hands-on 1: Preliminary installations

Your Turn!

Hands-on 1

ulhpc-tutorials.rtfd.io/en/latest/bigdata/#1-preliminary-installations

- **Reserve** an **interactive** job for the build
  - ↪ if possible within a `screen` session
- Install **Java** 7u80 and 8u152 — Step 1.a
- Install **Maven** 3.5.2 — Step 1.b
- Install Cmake 3.9.1, snappy 1.1.6 and protobuf 2.5.0 — Step 1.c

Hands-on 2: Install Hadoop

Your Turn!

Hands-on 2

ulhpc-tutorials.rtfd.io/en/latest/bigdata/#2-hadoop-installation

- We're going to install the most recent Hadoop by Cloudera
  - ↪ Grab the recipY/easyconfig `Hadoop-2.6.0-cdh5.12.0-native.eb`
  - ↪ Patch it
  - ↪ run the installation
- (**reminder**) installation is quite long
  - ↪ you **want** to have that done in a `screen` session...
    - ✓ **from the start** (i.e. on the access node)

What is a Distributed File System?

- Straightforward idea: **separate logical from physical storage.**
  - ↪ Not all files reside on a single physical disk,
  - ↪ or the same physical server,
  - ↪ or the same physical rack,
  - ↪ or the same geographical location,. . .
- **Distributed file system (DFS)**:
  - ↪ virtual file system that enables clients to access files
    - ✓ . . . as if they were stored locally.

What is a Distributed File System?

- Straightforward idea: **separate logical from physical storage.**
  - ↪ Not all files reside on a single physical disk,
  - ↪ or the same physical server,
  - ↪ or the same physical rack,
  - ↪ or the same geographical location,. . .
- **Distributed file system (DFS)**:
  - ↪ virtual file system that enables clients to access files
    - ✓ . . . as if they were stored locally.

- **Major DFS distributions**:
  - ↪ **NFS**: originally developed by Sun Microsystems, started in 1984
  - ↪ **AFS/CODA**: originally prototypes at Carnegie Mellon University
  - ↪ **GFS**: Google paper published in 2003, not available outside Google
  - ↪ **HDFS**: designed after GFS, part of Apache Hadoop since 2006

Distributed File System Architecture?

Master-Slave Pattern

- Single (or few) **master** nodes maintain state info. about clients
- All clients R&W requests go through the global master node.
- **Ex**: GFS, HDFS

Distributed File System Architecture?

Master-Slave Pattern

- Single (or few) **master** nodes maintain state info. about clients
- All clients R&W requests go through the global master node.
- **Ex**: GFS, HDFS

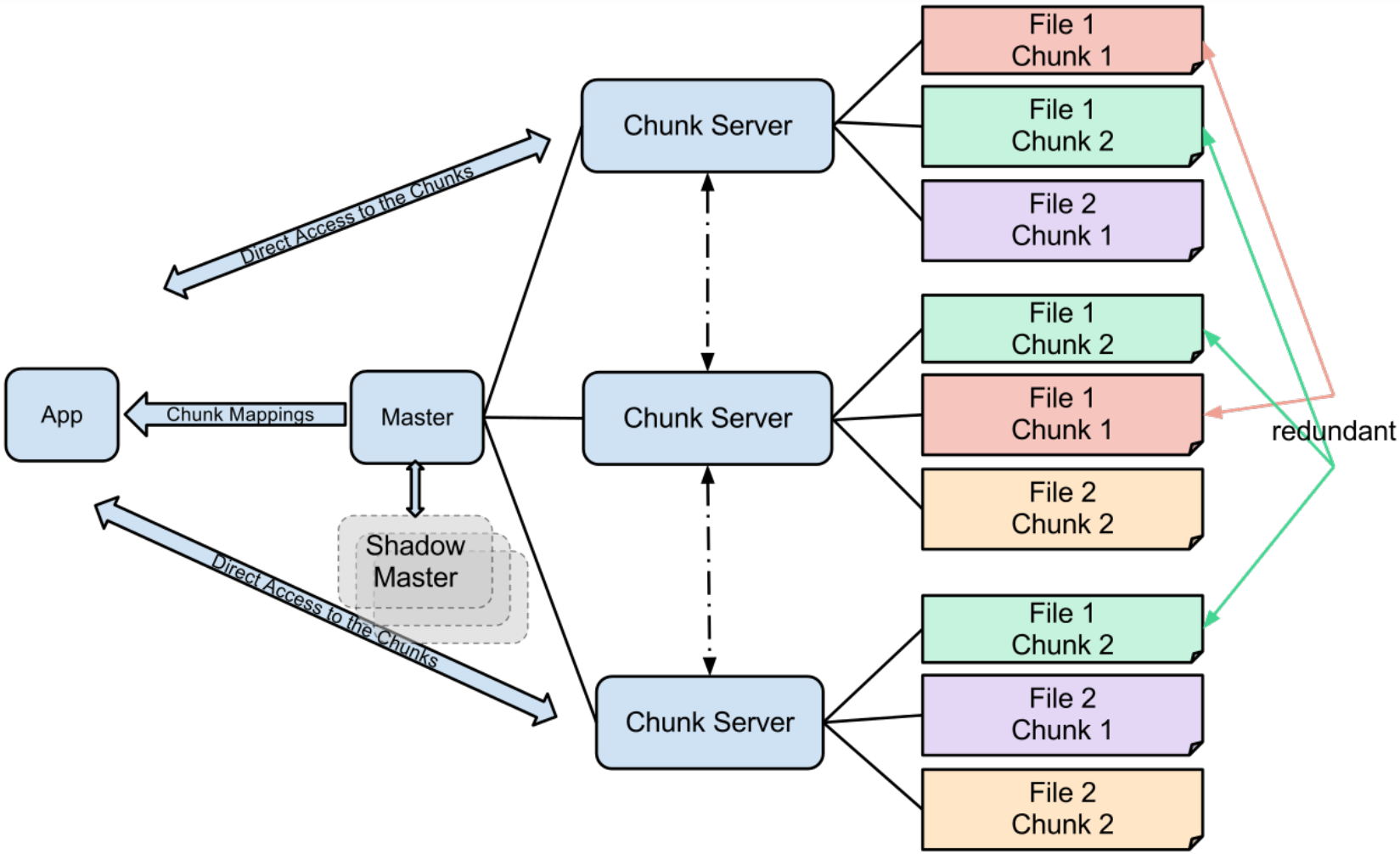
Peer-to-Peer Pattern

- No global state information.
- Each node may both serve and process data.

Google File System (GFS) (2003)

- Radically different architecture compared to NFS, AFS and CODA.
  - ↪ specifically tailored towards **large-scale** and **long-running analytical processing tasks**
  - ↪ over thousands of storage nodes.
- **Basic assumption**:
  - ↪ client nodes (aka. *chunk servers*) may fail any time!
  - ↪ Bugs or hardware failures.
  - ↪ Special tools for monitoring, periodic checks.
  - ↪ Large files (multiple GBs or even TBs) are split into 64 MB *chunks*.
  - ↪ Data modifications are mostly append operations to files.
  - ↪ Even the master node may fail any time!
    - ✓ Additional *shadow master* fallback with read-only data access.
- Two types of reads: Large sequential reads & small random reads

Google File System (GFS) (2003)



GFS Consistency Model

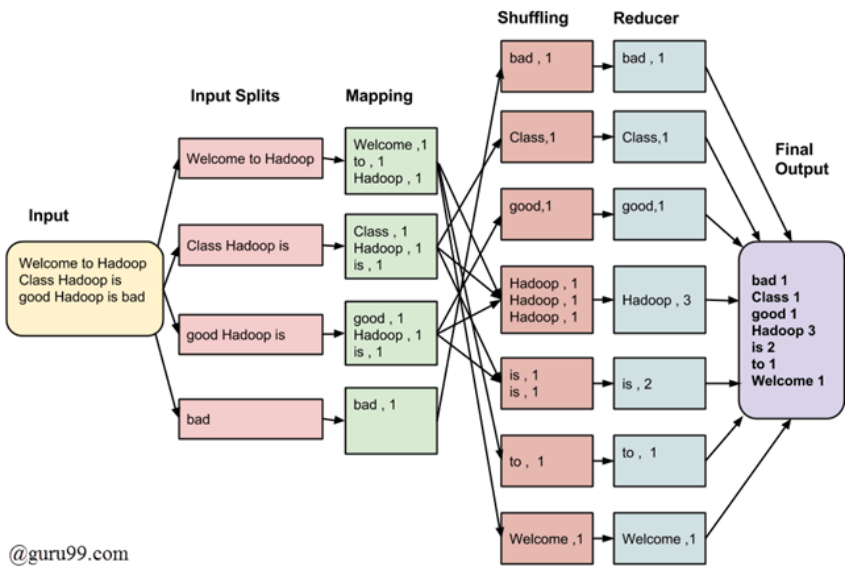
- **Atomic File Namespace Mutations**
  - ↪ File creations/deletions centrally controlled by the master node.
  - ↪ Clients typically create and write entire file,
    - ✓ then add the file name to the file namespace stored at the master.
- **Atomic Data Mutations**
  - ↪ only 1 atomic modification of 1 replica (!) at a time is guaranteed.
- **Stateful Master**
  - ↪ Master sends regular **heartbeat** messages to the chunk servers
  - ↪ Master keeps chunk locations of all files (+ replicas) in memory.
  - ↪ locations not stored persistently...
    - ✓ but polled from the clients at startup.
- **Session Semantics**
  - ↪ Weak consistency model for file replicas and client caches only.
  - ↪ Multiple clients may read and/or write the same file concurrently.
  - ↪ The client that last writes to a file **wins**.

Fault Tolerance & Fault Detection

- **Fast Recovery**
  - ↪ master & chunk servers can restore their states and (re-)start in s.
    - ✓ regardless of previous termination conditions.
- **Master Replication**
  - ↪ *shadow master* provides RO access when primary master is down.
    - ✓ Switches back to read/write mode when primary master is back.
  - ↪ Master node does not keep a persistent state info. of its clients,
    - ✓ rather polls clients for their states when started.
- **Chunk Replication & Integrity Checks**
  - ↪ chunk divided into 64 KB blocks, each with its own 32-bit checksum
    - ✓ verified at read and write times.
  - ↪ Higher replication factors for more intensively requested chunks (**hotspots**) can be configured.

Map-Reduce

- Breaks the processing into two main phases:
  1. the **map** phase
  2. the **reduce** phase.
- Each phase has key-value pairs as input and output,
  - ↪ the types of which may be chosen by the programmer.
  - ↪ the programmer also specifies the `map` and `reduce` functions



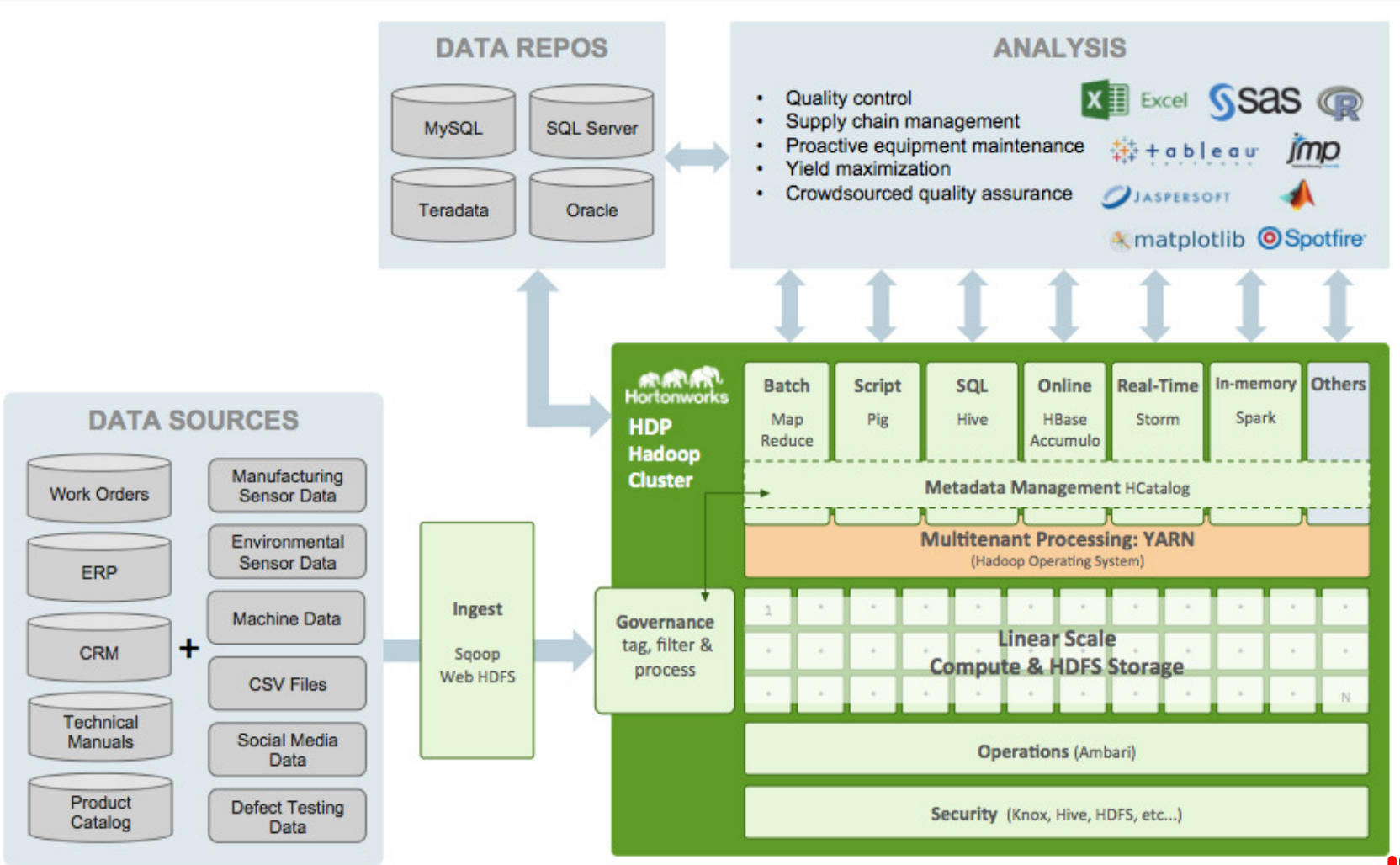
Hadoop

- Initially started as a student project at Yahoo! labs in 2006
  - ↪ Open-source Java implem. of GFS and MapReduce frameworks
- Switched to Apache in 2009. Now consists of three main modules:
  1. **HDFS**: Hadoop distributed file system
  2. **YARN**: Hadoop job scheduling and resource allocation
  3. **MapReduce**: Hadoop adaptation of the MapReduce principle

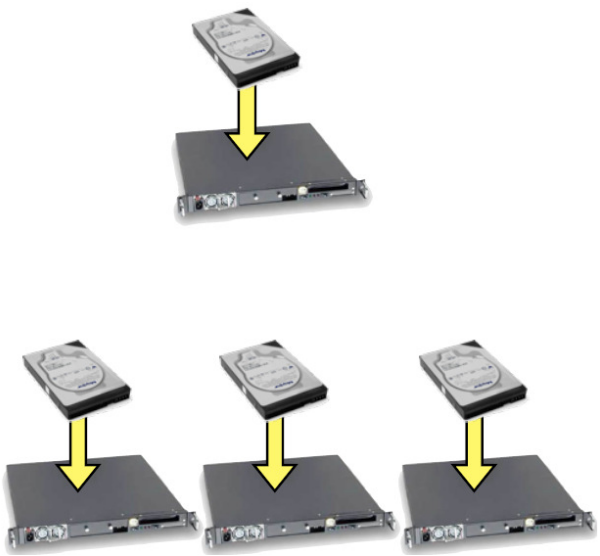
Hadoop

- Initially started as a student project at Yahoo! labs in 2006
  - ↪ Open-source Java implem. of GFS and MapReduce frameworks
- Switched to Apache in 2009. Now consists of three main modules:
  1. **HDFS**: Hadoop distributed file system
  2. **YARN**: Hadoop job scheduling and resource allocation
  3. **MapReduce**: Hadoop adaptation of the MapReduce principle
- Basis for many other open-source Apache toolkits:
  - ↪ **PIG/PigLatin**: file-oriented data storage & script-based query language
  - ↪ **HIVE**: distributed SQL-style data warehouse
  - ↪ **HBase**: distributed key-value store
  - ↪ **Cassandra**: fault-tolerant distributed database, etc.
- HDFS still mostly follows the original GFS architecture.

Hadoop Ecosystem

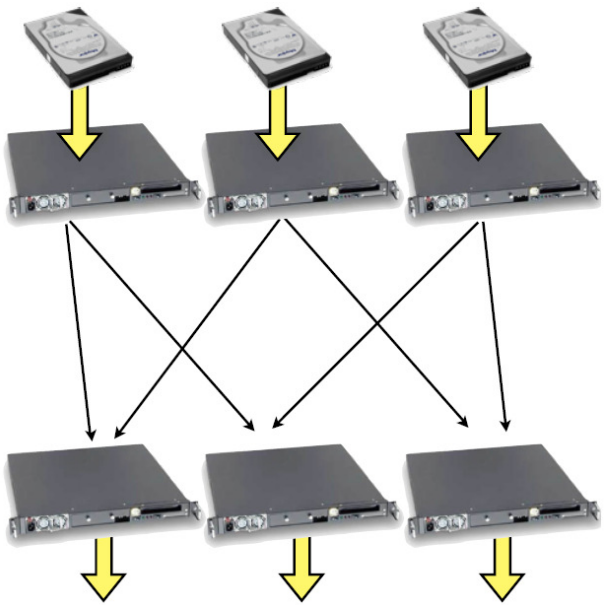


Scale-Out Design



- HDD streaming speed ~ 50MB/s
  - ↪ 3TB =17.5 hrs
  - ↪ 1PB = 8 months
- Scale-out (weak scaling)
  - ↪ **FS distributes data on ingest**
- Seeking too slow
  - ↪ ~10ms for a seek
  - ↪ Enough time to read half a megabyte
- **Batch processing**
- Go through entire data set in one (or small number) of passes

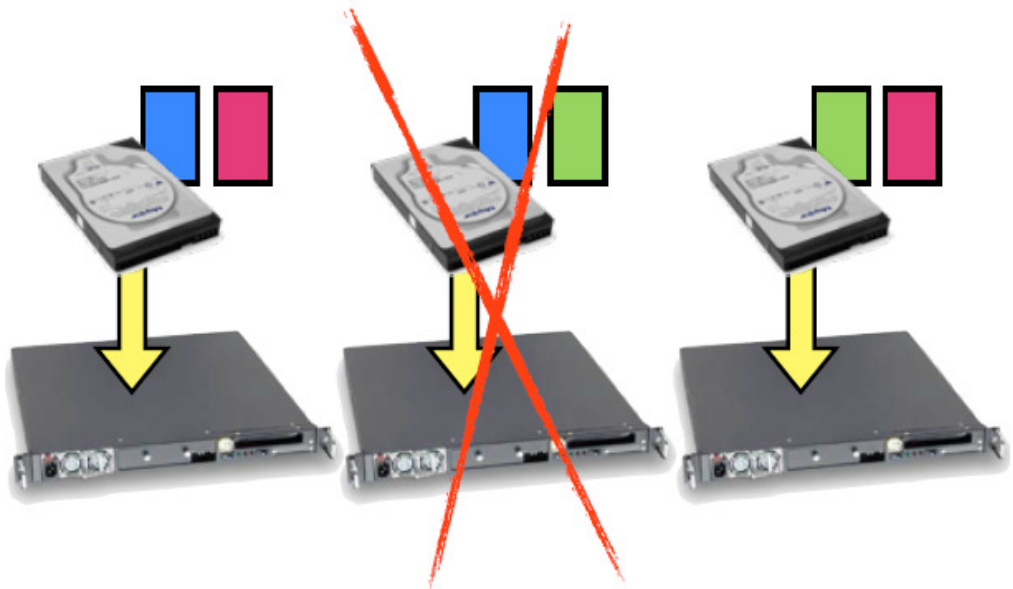
Combining Results



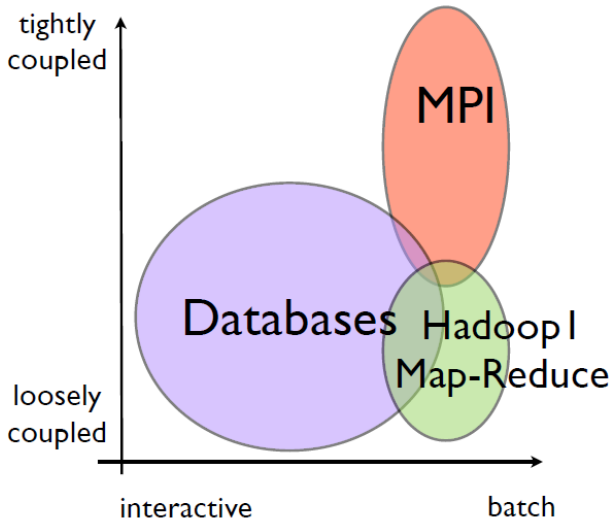
- Each node preprocesses its local data
  - ↪ Shuffles its data to a small number of other nodes
- Final processing, output is done there

Fault Tolerance

- Data also replicated upon ingest
- Runtime watches for dead tasks, restarts them on live nodes
- Re-replicates

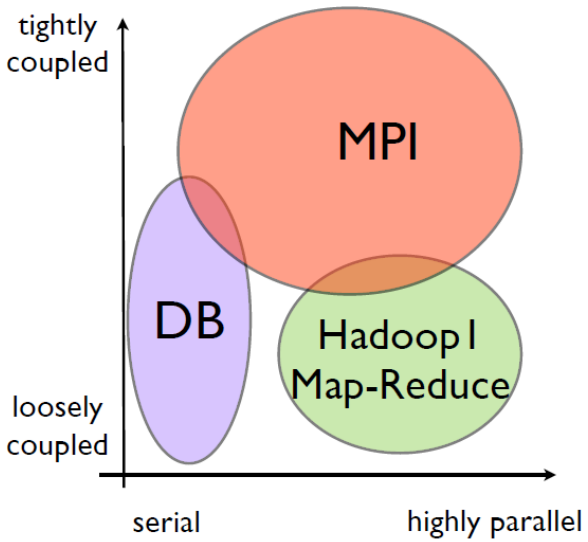


Hadoop: What is it Good At?



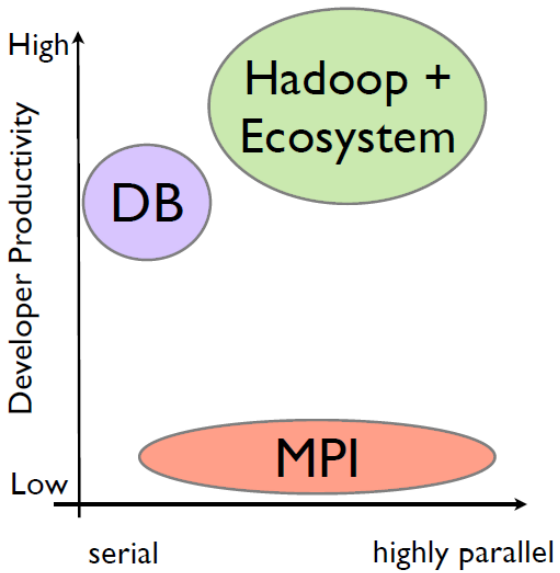
- **Classic Hadoop 1.x is all about batch processing** of massive amounts of data
  - ↪ Not much point below ~1TB
- Map-Reduce is relatively loosely coupled;
  - ↪ one **shuffle** phase.
- Very strong weak scaling in this model
  - ↪ more data, more nodes.
- **Batch**:
  - ↪ process all data in one go
    - ✓ w/classic Map Reduce
  - ↪ Current Hadoop has many other capabilities besides batch - more later

Hadoop: What is it Good At?



- **Compare with databases**
  - ↪ very good at working on small subsets of large databases
    - ✓ DBs: very interactive for many tasks
    - ✓ . . . yet have been difficult to scale
- **Compare with HPC (MPI)**
  - ↪ Also typically batch
  - ↪ Can (and does) go up to enormous scales
- Works extremely well for very tightly coupled problems:
  - ↪ zillions of iterations/timesteps/ exchanges.

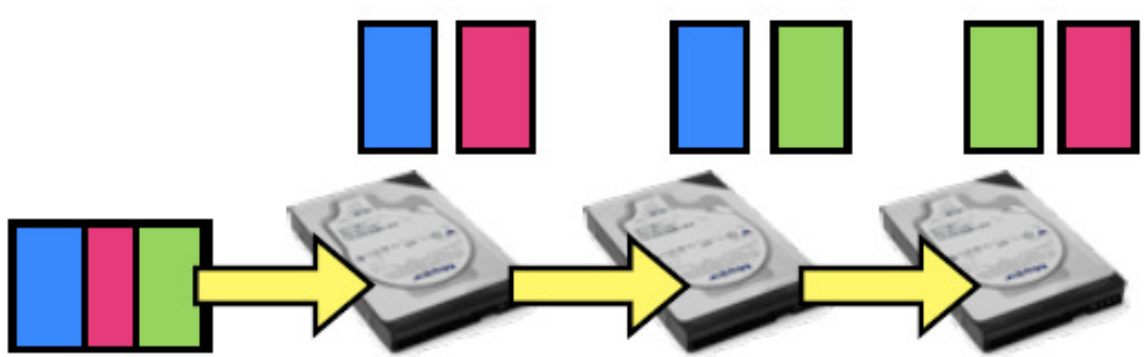
Hadoop vs HPC



- We HPC users might be tempted to an unseemly smugness
  - ↪ *They solved the problem of disk-limited, loosely-coupled, data analysis by throwing more disks at it and weak scaling? Ooooooooh*
- **We would be wrong**.
  - ↪ A single novice developer can write:
    - ✓ real, scalable,
    - ✓ 1000+ node data-processing tasks in Hadoop-family tools in an afternoon.
  - ↪ In MPI... less likely...

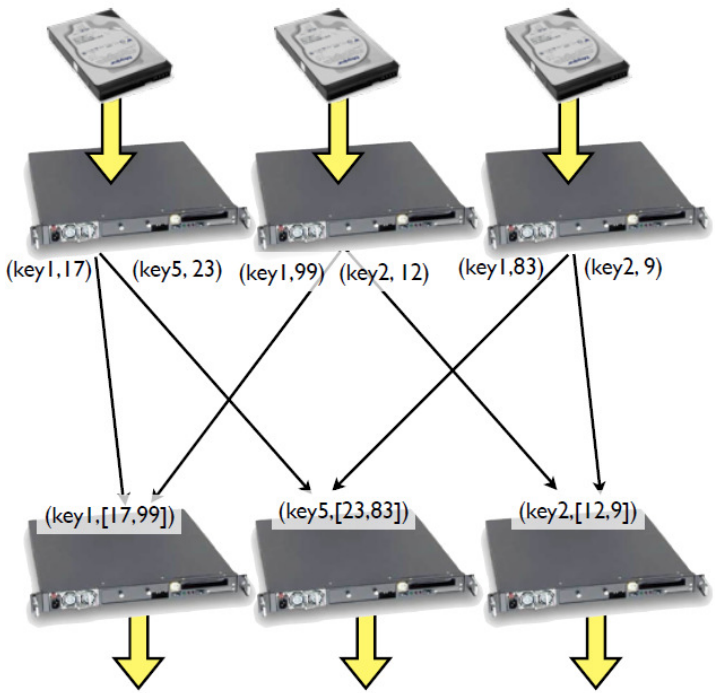
Data Distribution: Disk

- Hadoop & al. arch. handle the hardest part of parallelism for you
  - ↪ aka **data distribution**.
- **On disk**:
  - ↪ HDFS distributes, replicates data as it comes in
  - ↪ Keeps track of computations local to data



Data Distribution: Network

- **On network**: **Map Reduce** (eg) works in terms of key-value pairs.
  - ↪ Preprocessing (`map`) phase ingests data, emits (k, v) pairs
  - ↪ Shuffle phase assigns reducers,
    - ✓ gets all pairs with same key onto that reducer.
  - ↪ Programmer does not have to design communication patterns



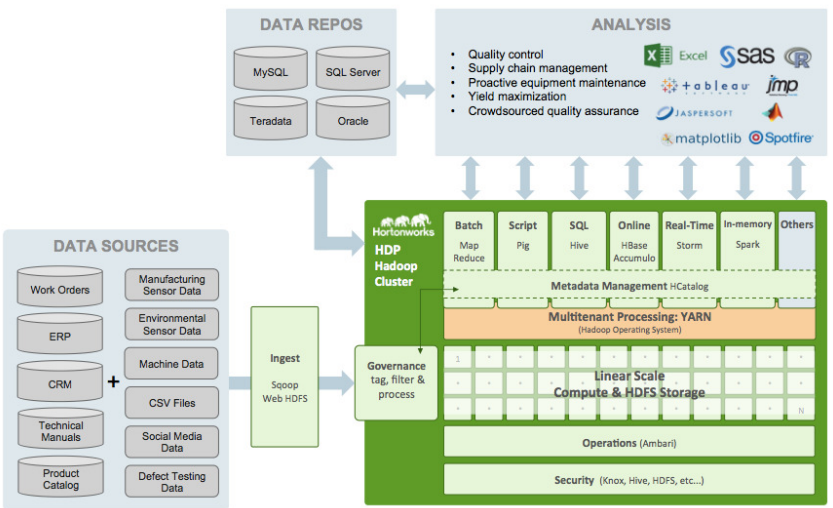
Makes the problem easier

- **Hardest parts of parallel programming with HPC tools**
  - ↪ Decomposing the problem, and,
  - ↪ Getting the intermediate data where it needs to go,

- **Hadoop does that for you**
  - ↪ automatically
  - ↪ for a wide range of problems.

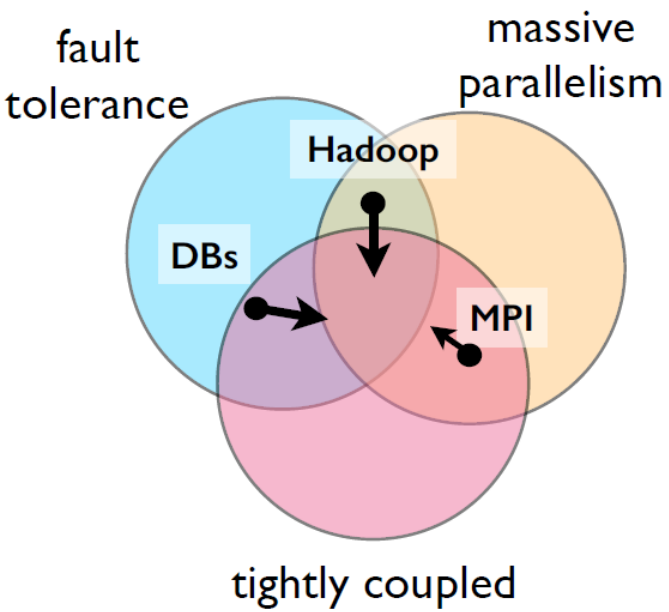
Built a reusable substrate

- HDFS and the MapReduce layer were very well architected.
  - ↪ Enables many higher-level tools
  - ↪ Data analysis, machine learning, NoSQL DBs,. . .
- Extremely productive environment
  - ↪ And Hadoop 2.x (YARN) is now much much more than just MapReduce



Hadoop and HPC

- **Not either-or anyway**
  - ↪ Use HPC to generate big / many simulations,
  - ↪ Use Hadoop to analyze results
    - ✓ **Ex**: Use Hadoop to preprocess huge input data sets (ETL),
    - ✓ ... and HPC to do the tightly coupled computation afterwards.
- **In all cases**: Everything is Converging



The Hadoop Filesystem

- **HDFS is a distributed parallel filesystem**
  - ↪ Not a general purpose file system
    - ✓ does not implement posix
    - ✓ cannot just mount it and view files
- Access via `hdfs fs` commands or programatic APIs
- Security slowly improving

```
$> hdfs fs -[cmd]
```

cat	chgrp
chmod	chown
copyFromLocal	copyToLocal
cp	du
dus	expunge
get	getmerge
ls	lsr
mkdir	movefromLocal
mv	put
rm	rmr
setrep	stat
tail	test
text	touchz

The Hadoop Filesystem

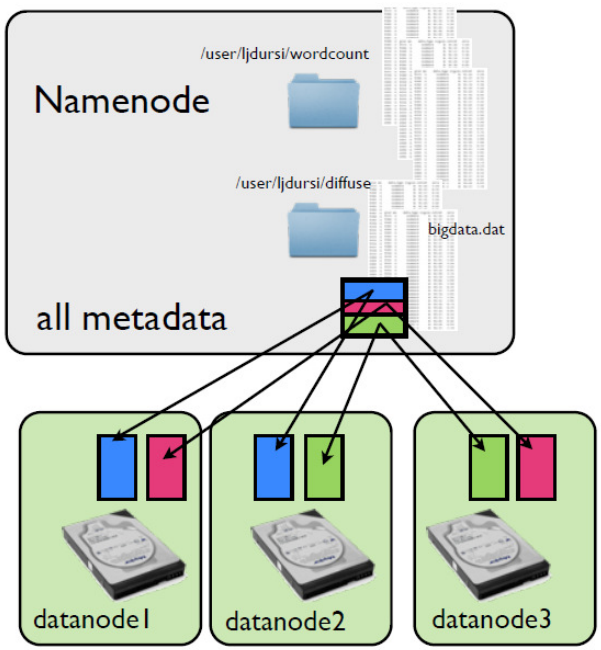
- **Required** to be:
  - ↪ able to deal with large files, large amounts of data
  - ↪ scalable & reliable in the presence of failures
  - ↪ fast at reading contiguous streams of data
  - ↪ only need to write to new files or append to files
  - ↪ require only commodity hardware
- **As a result**:
  - ↪ Replication
  - ↪ Supports mainly high bandwidth, **not** especially low latency
  - ↪ No caching
    - ✓ what is the point if primarily for streaming reads?
    - ✓ Poor support for seeking around files
    - ✓ Poor support for zillions of files
  - ↪ Have to use separate API to see filesystem
  - ↪ Modelled after Google File System (2004 Map Reduce paper)

Hadoop vs HPC

- HDFS is a **block-based** FS
  - ↪ A file is broken into blocks,
  - ↪ these blocks are distributed across nodes
- **Blocks are large**;
  - ↪ 64MB is default,
  - ↪ many installations use 128MB or larger
- Large block size
  - ↪ time to stream a block much larger than time disk time to access the block.

```
# Lists all blocks in all files:
$> hdfs fsck / -files -blocks
```

Datanodes and Namenode

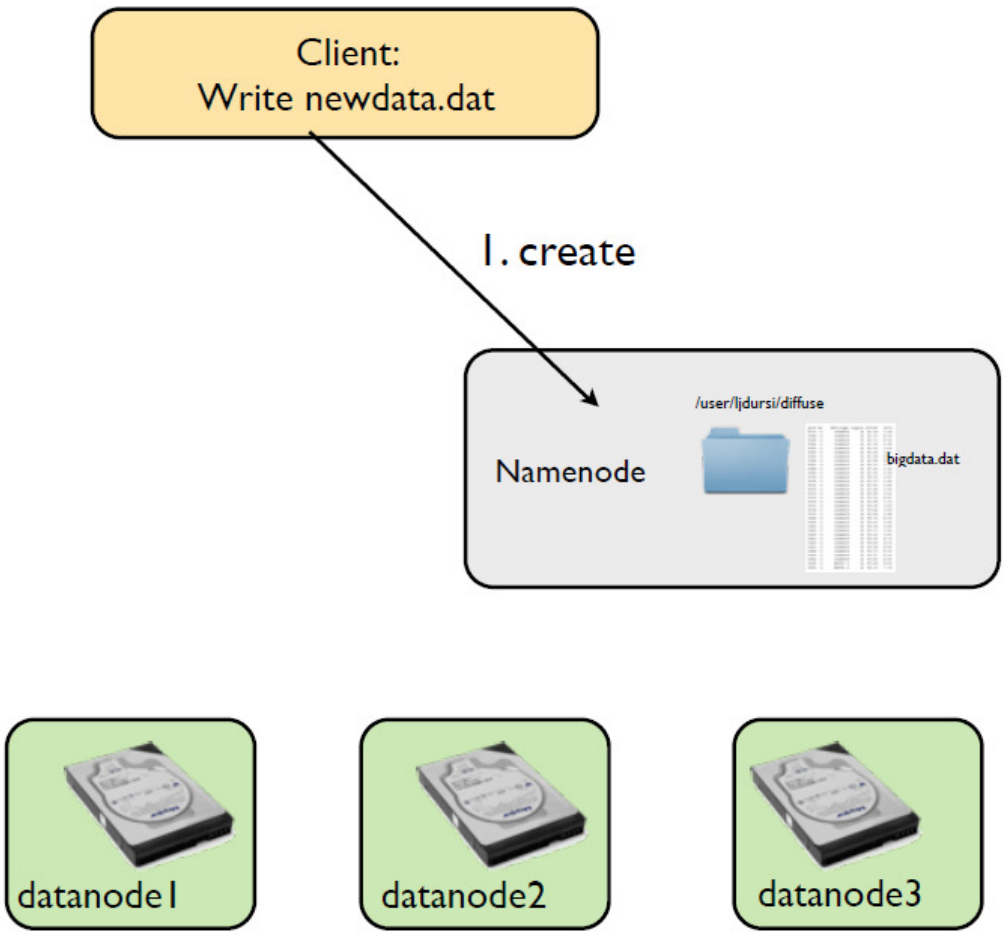


Two types of nodes in the filesystem:

1. **Namenode**
   - ↪ stores all metadata / block locations in memory
   - ↪ Metadata updates stored to persistent journal
2. **Datanodes**
   - ↪ store/retrieve blocks for client/namenode

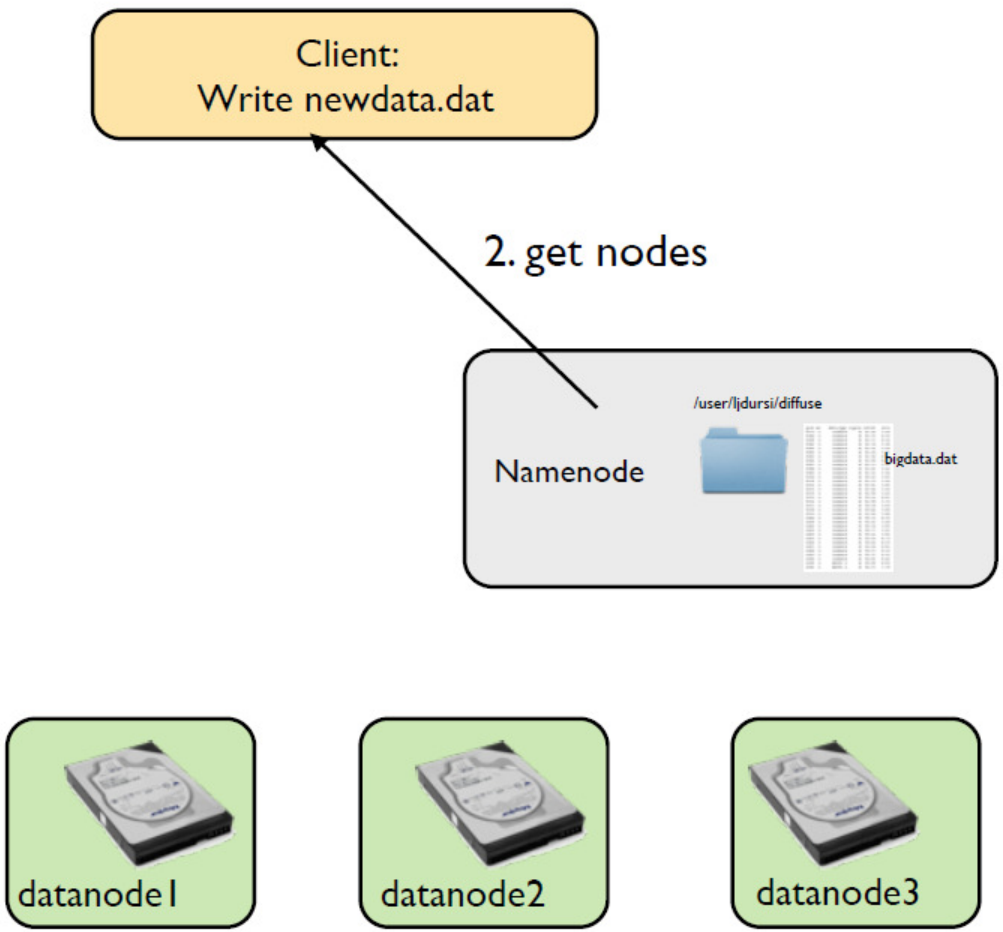
- Newer versions of Hadoop: federation
  - ↪ $\neq$ namenodes for `/user`, `/data`...
  - ↪ High Availability namenode pairs

Writing a file



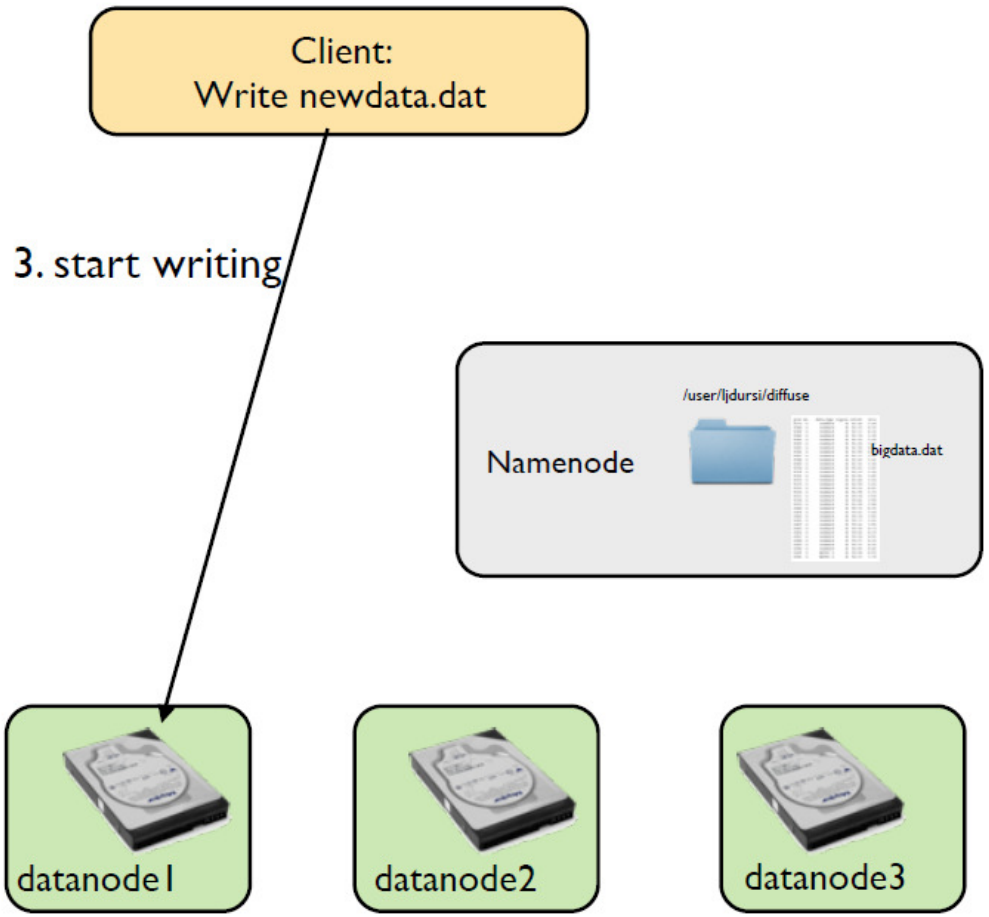
- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back
  - ↪ Complete

Writing a file



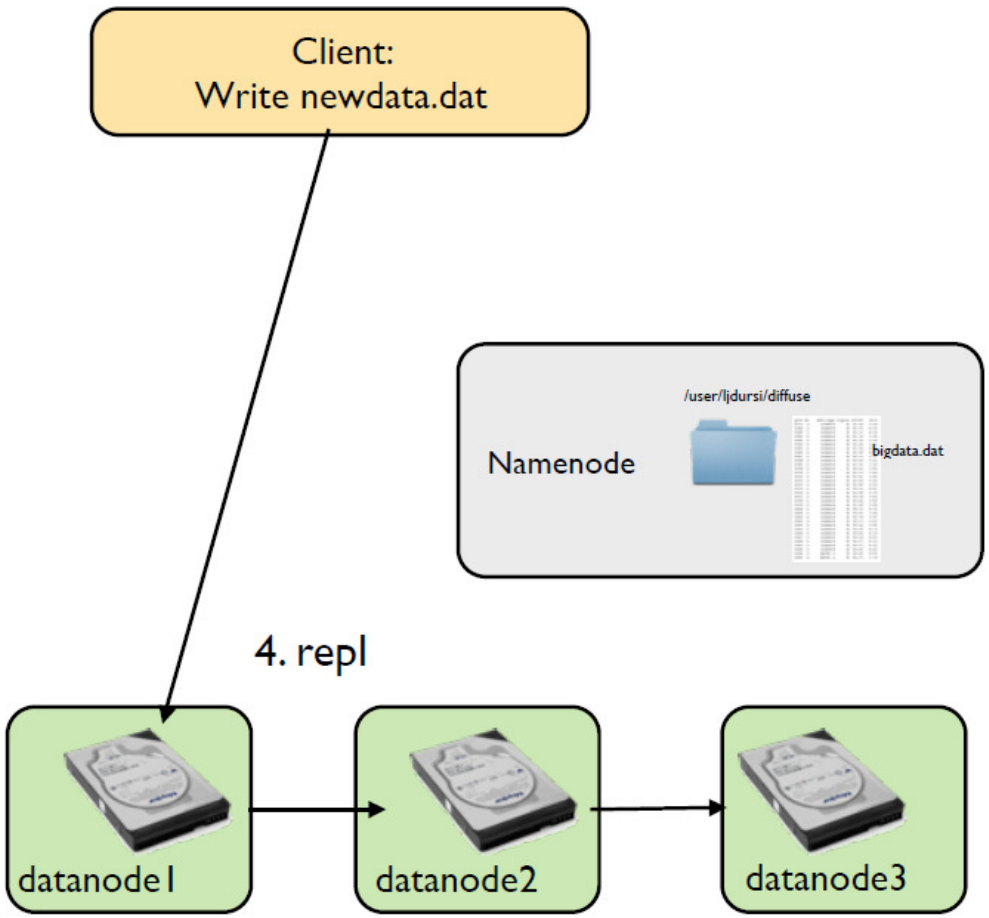
- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back
  - ↪ Complete

Writing a file



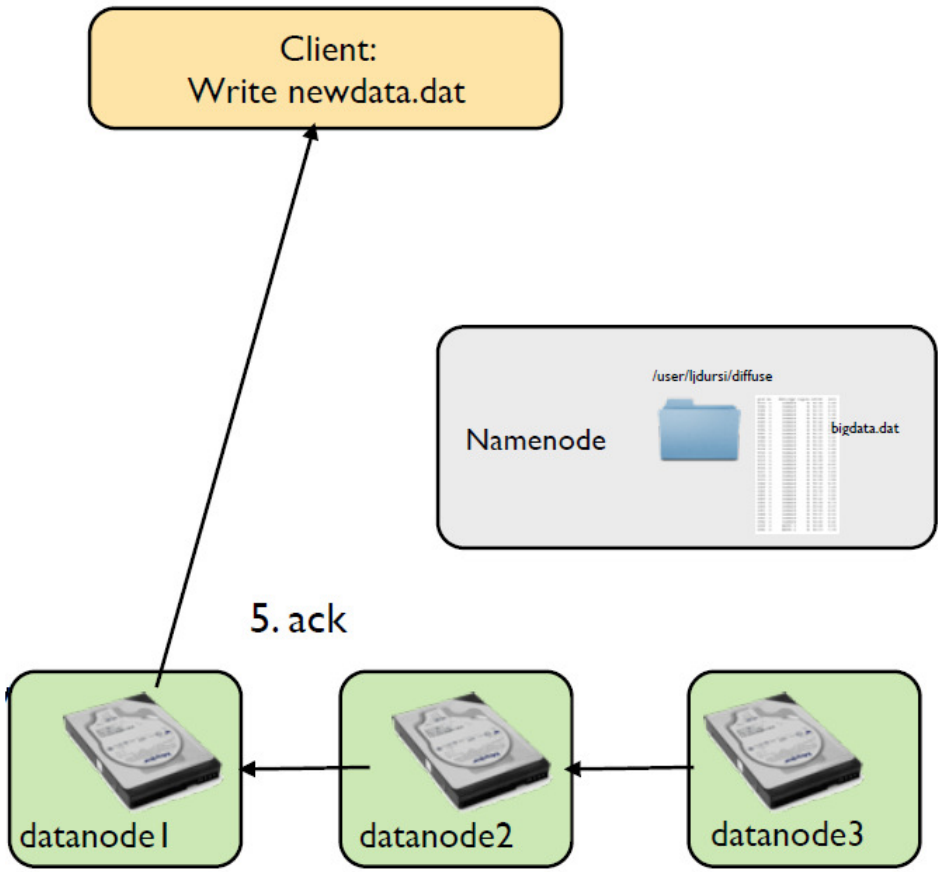
- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back
  - ↪ Complete

Writing a file



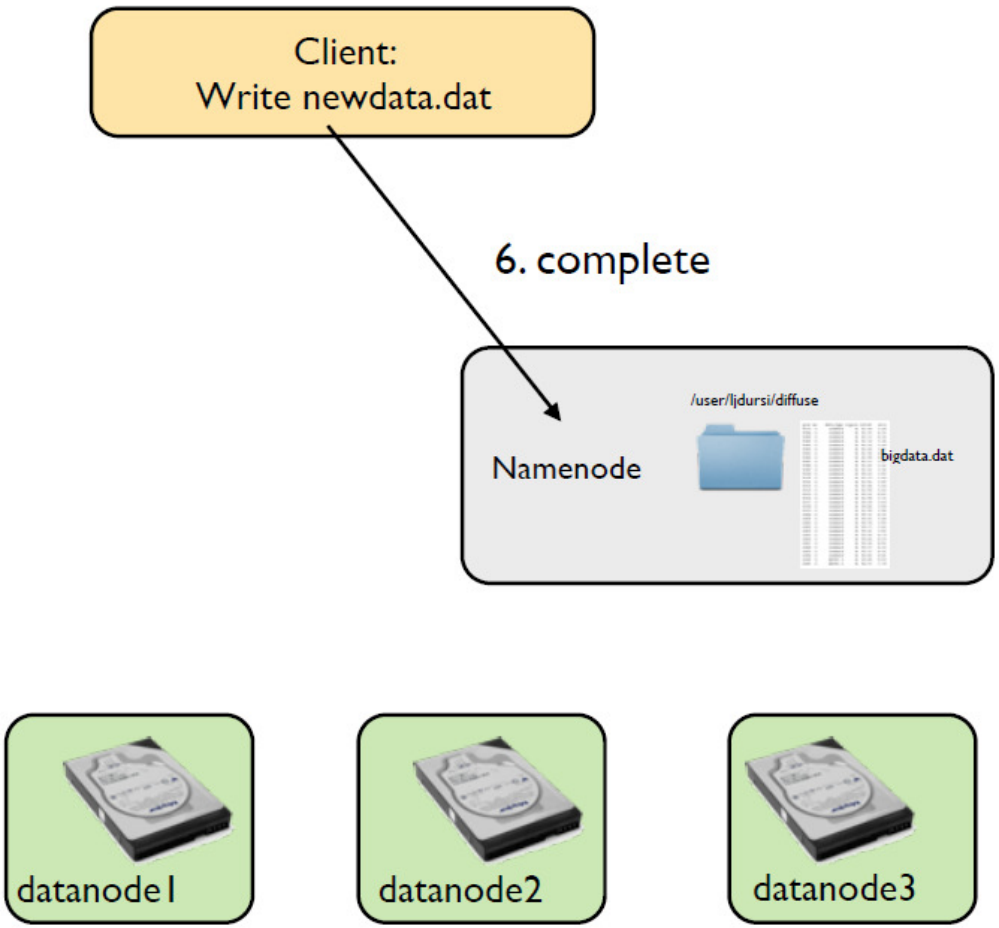
- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back
  - ↪ Complete

Writing a file



- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back **(while writing)**
  - ↪ Complete

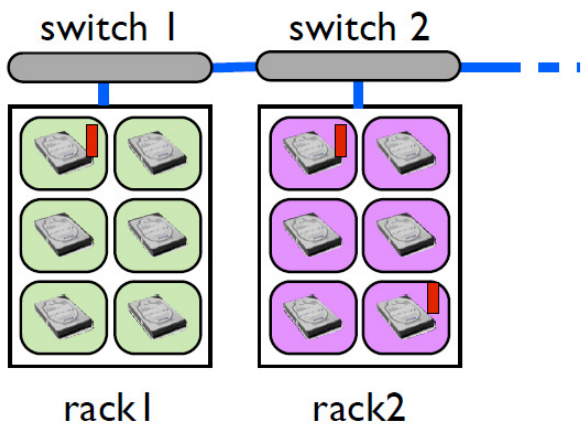
Writing a file



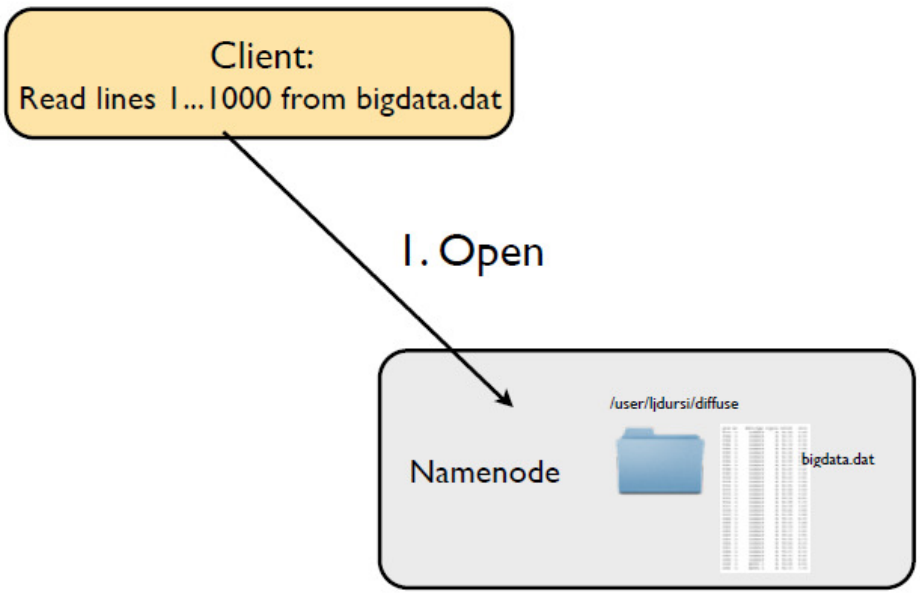
- **Writing a file** multiple stage process:
  - ↪ Create file
  - ↪ Get nodes for blocks
  - ↪ Start writing
  - ↪ Data nodes coordinate replication
  - ↪ Get ack back **(while writing)**
  - ↪ Complete

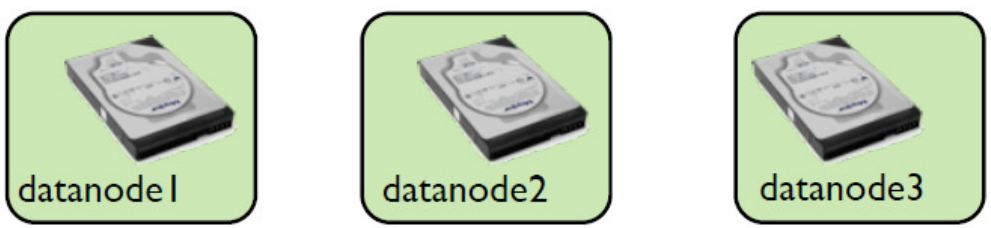
Where to Replicate?

- **Tradeoff** to choosing replication locations
  - ↪ **Close**: faster updates, less network bandwidth
  - ↪ **Further**: better failure tolerance
- **Default strategy**:
  1. copy on different location on same node
  2. second on different *rack*(switch),
  3. third on same rack location, different node.
- Strategy configurable.
  - ↪ Need to configure Hadoop file system to know location of nodes



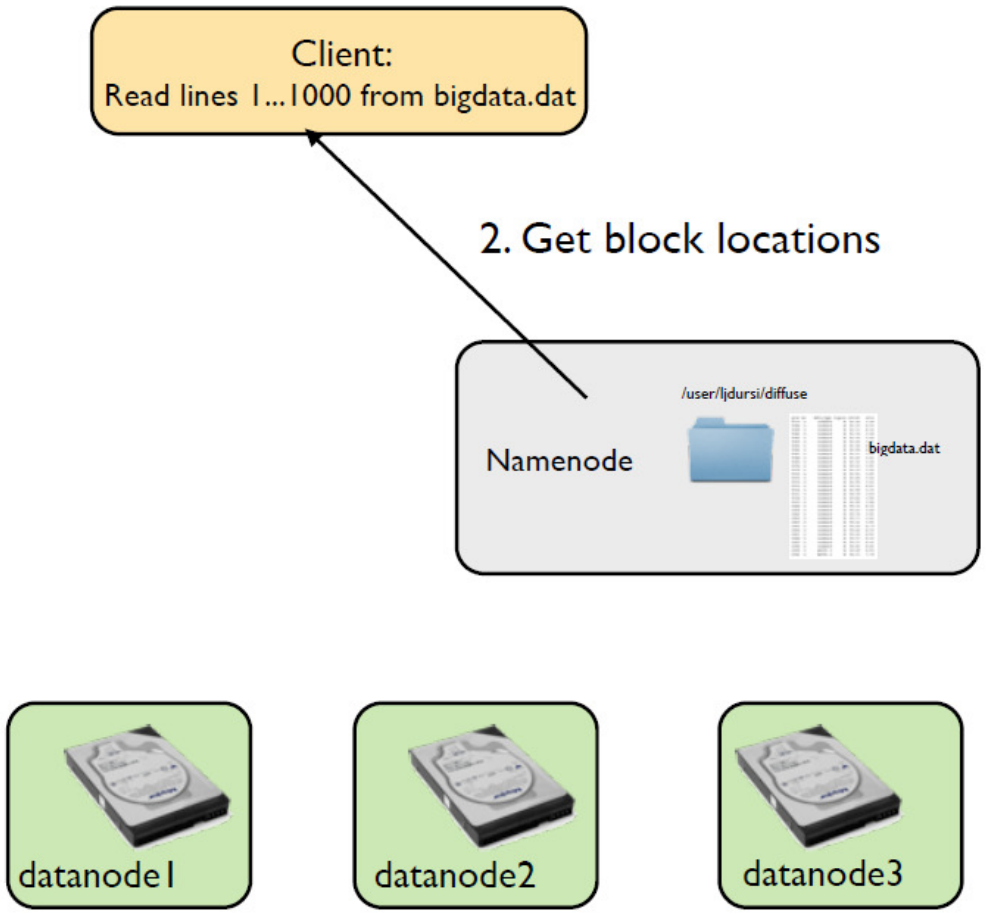
Reading a file





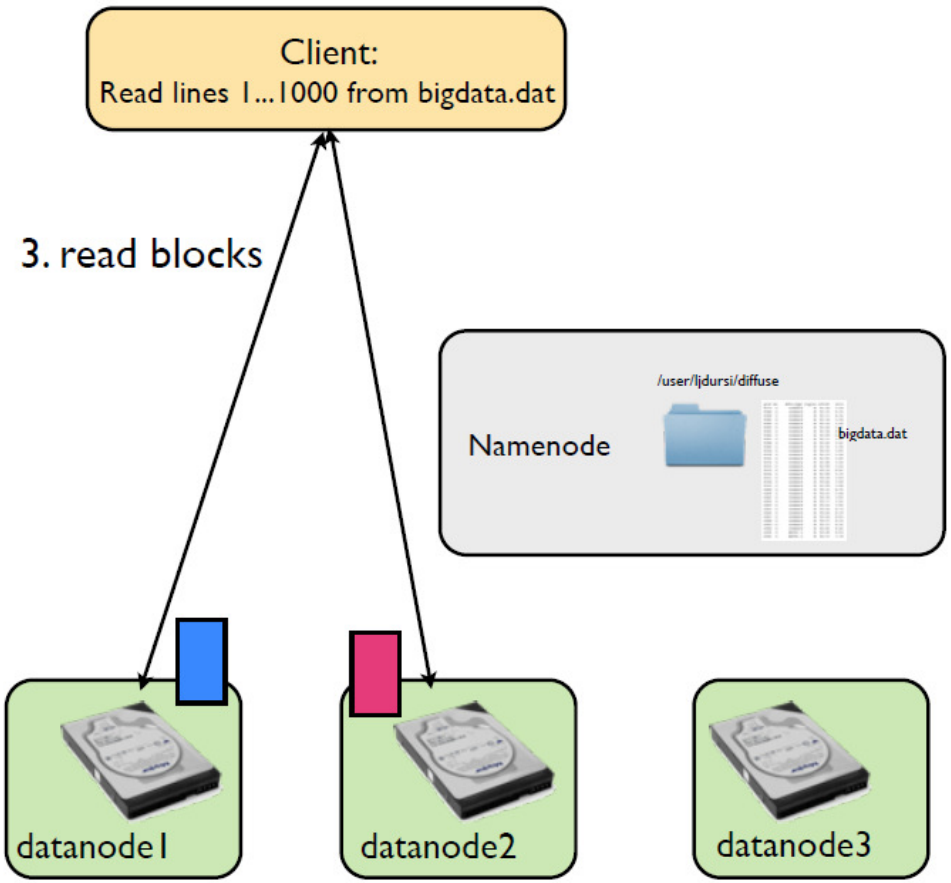
- **Reading** a file
  - ↪ Open call
  - ↪ Get block locations
  - ↪ Read from a replica

Reading a file



- **Reading** a file
  - ↪ Open call
  - ↪ Get block locations
  - ↪ Read from a replica

Reading a file



- **Reading** a file
  - ↪ Open call
  - ↪ Get block locations
  - ↪ Read from a replica

Configuring HDFS

- Need to tell HDFS how to set up filesystem
  - ↪ `data.dir`, `name.dir`
    - ✓ where on local system (eg, local disk) to write data
  - ↪ parameters like replication
    - ✓ how many copies to make
  - ↪ default name - default file system to use
  - ↪ Can specify multiple FSs

Configuring HDFS

```
<!-- $HADOOP_PREFIX/etc/hadoop/core-site.xml -->
<configuration>
  <property>
   <name>fs.defaultFS</name>
   <value>hdfs://<server>:9000</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>/home/username/hdfs/data</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>/home/username/hdfs/name</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
</configuration>
```

Configuring HDFS

- In Practice, in `single` mode
  - ↪ Only one node to be used, the VM
  - ↪ **default server**: `localhost`
  - ↪ Since only one node:
    - ✓ need to specify replication factor of 1, or will always fail

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://localhost:9000</value>
</property>
[...]
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
```

Configuring HDFS

- You will need to make sure that environment variables are set
  - ↪ path to Java, path to Hadoop...
  - ↪ Easybuild does **most** of the job for you
- You will need passwordless SSH access accross all nodes
- You can then start processes on various FS nodes

Configuring HDFS

- You will need to make sure that environment variables are set
  - ↪ path to Java, path to Hadoop...
  - ↪ Easybuild does **most** of the job for you
- You will need passwordless SSH access accross all nodes
- You can then start processes on various FS nodes

- Once configuration files are set up,
  - ↪ you can format the namenode like so
  - ↪ you can start up just the file systems

```
$> hdfs namenode -format
$> start-dfs.sh
```

Using HDFS

- Once the file system is up and running,
  - ↪ ... you can copy files back and forth

```
$> hadoop fs -{get|put|copyFromLocal|copyToLocal} [...]
```

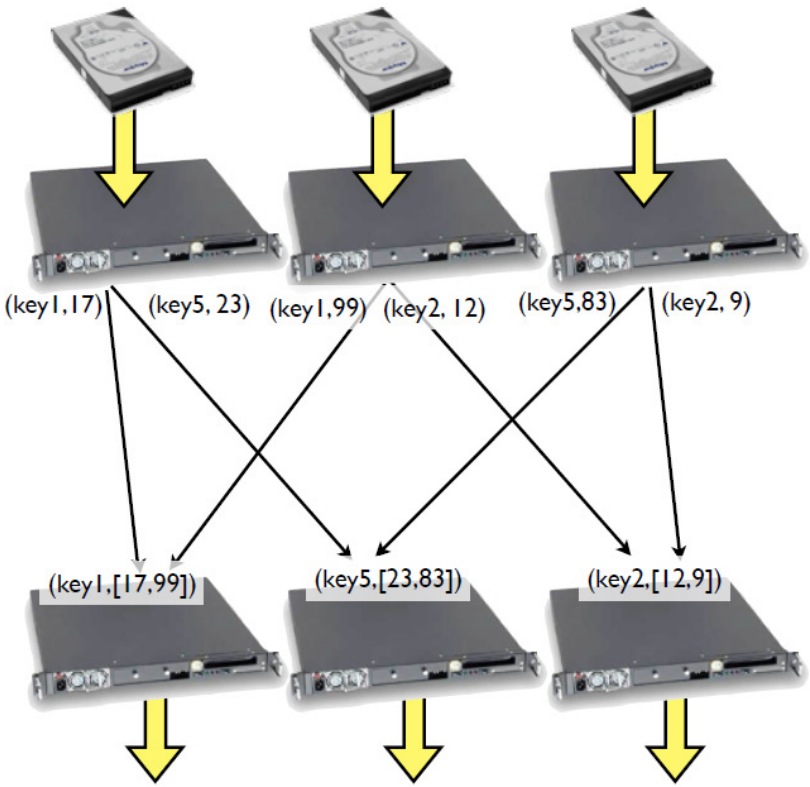
- Default directory is /user/${username}
  - ↪ **Nothing like a cd**

```
$> hdfs fs -mkdir /home/vagrant/hdfs-test
$> hdfs fs -ls    /home/vagrant
$> hdfs fs -ls    /home/vagrant/hdfs-test
$> hdfs fs -put data.dat /home/vagrant/hdfs-test
$> hdfs fs -ls    /home/vagrant/hdfs-test
```

Using HDFS

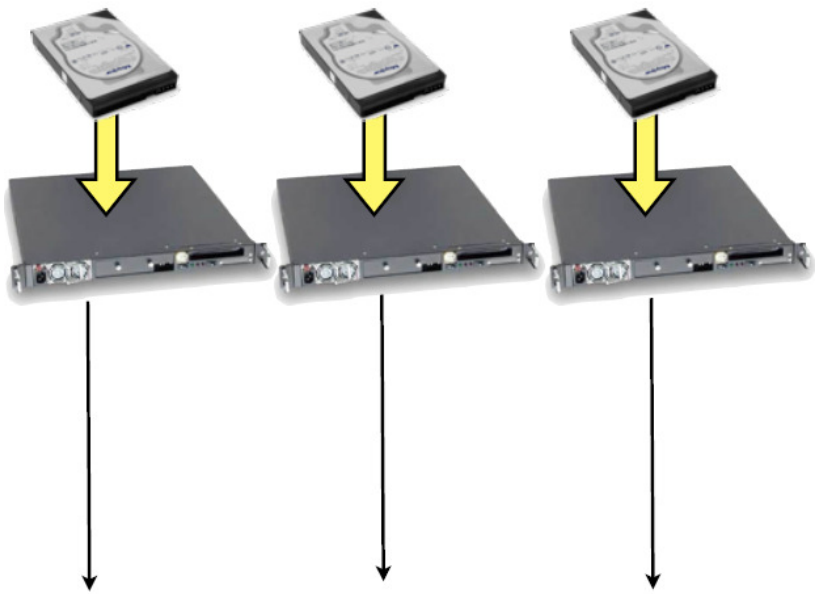
- In general, the data files you send to HDFS will be **large**
  - ↪ or else why bother with Hadoop.
- Do not want to be constantly copying back and forth
  - ↪ **view, append *in place***
- Several APIs to accessing the HDFS
  - ↪ Java, C++, Python
- Here, we use one to get a file status, and read some data from it at some given offset

Back to Map-Reduce



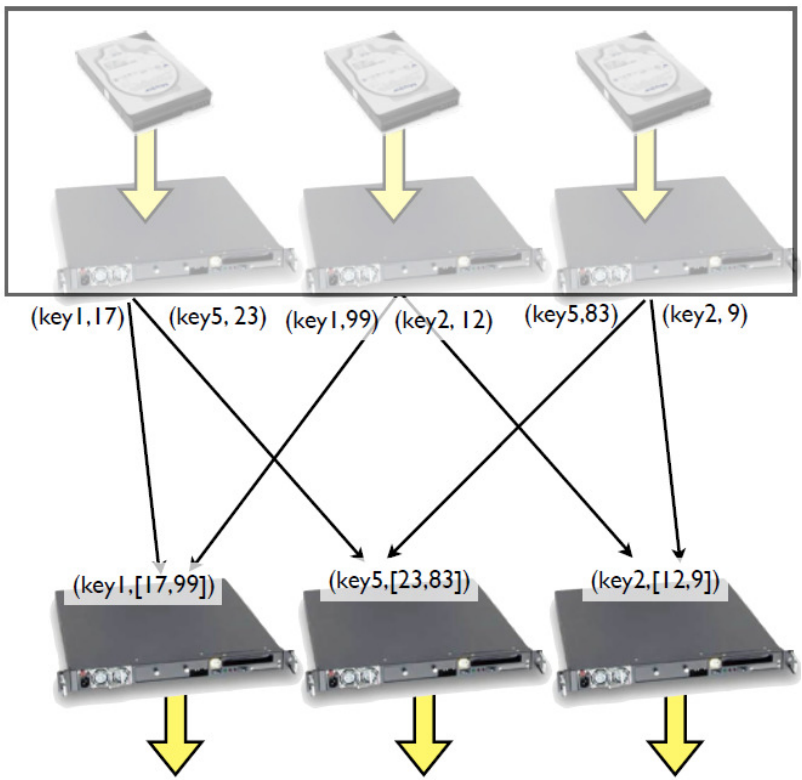
- Map processes **one element at a time**
  - ↪ emits results as (key, value) pairs.
- All results with **same key are gathered to the same reducers**
  - ↪ Reducers process list of values
  - ↪ emit results as (key, value) pairs

Map



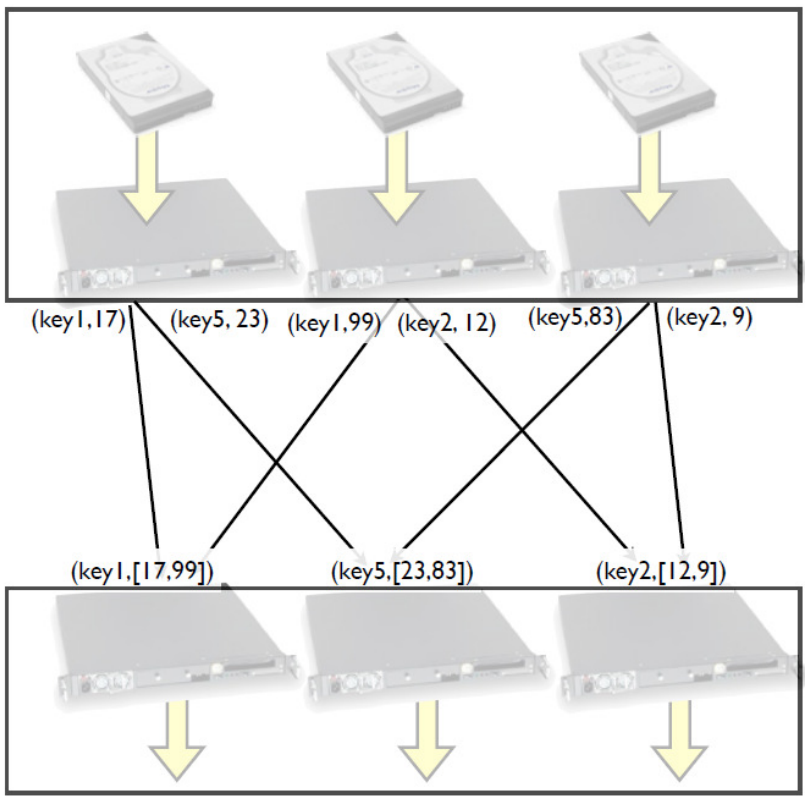
- All coupling done during **shuffle** phase
  - ↪ Embarrassingly parallel task
  - ↪ all map
- Take input, map it to output, done.
- **Famous case**
  - ↪ NYT using Hadoop to convert 11 million image files to PDFs
    - ✓ almost pure serial farm job

Reduce



- Reducing gives the coupling
- In the case of the NYT task:
  - ↪ not quite embarrassingly parallel:
    - ✓ images from multi-page articles
    - ✓ Convert a page at a time,
    - ✓ gather images with same article id onto node for conversion

Shuffle

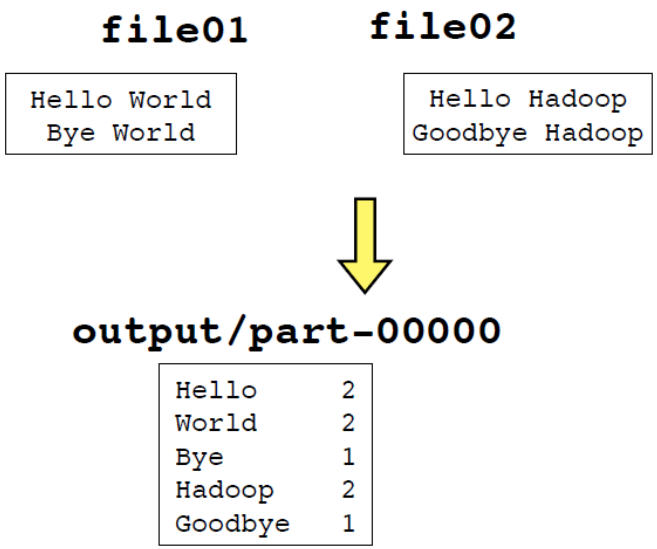


- **shuffle is part of the Hadoop magic**
  - ↪ By default, keys are hashed
  - ↪ hash space is partitioned between reducers
- On **reducer**:
  - ↪ gathered (k,v) pairs from mappers are sorted by key,
  - ↪ then merged together by key
  - ↪ Reducer then runs on one (k,[v]) tuple at a time
- you can supply your own partitioner
  - ↪ Assign **similar** keys to same node
  - ↪ Reducer still only sees one (k, [v]) tuple at a time.

Example: Wordcount

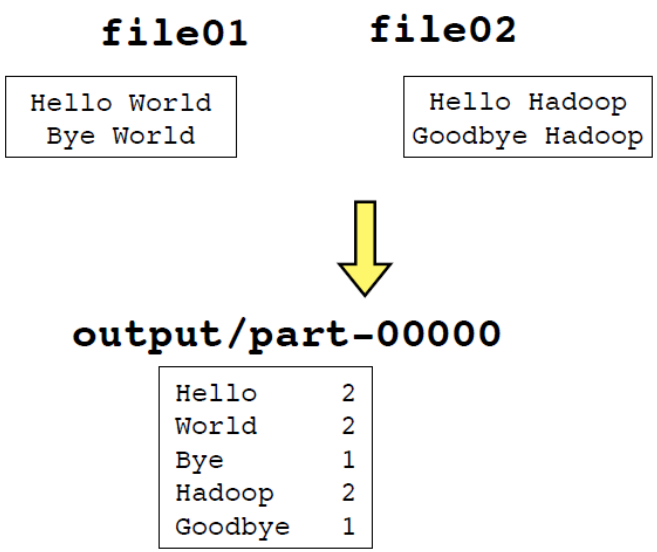
- Was used as an example in the original MapReduce paper
  - ↪ Now basically the **hello world** of map reduce

- **Problem description**: Given a **set** of documents:
  - ↪ **count occurences of words** within these documents

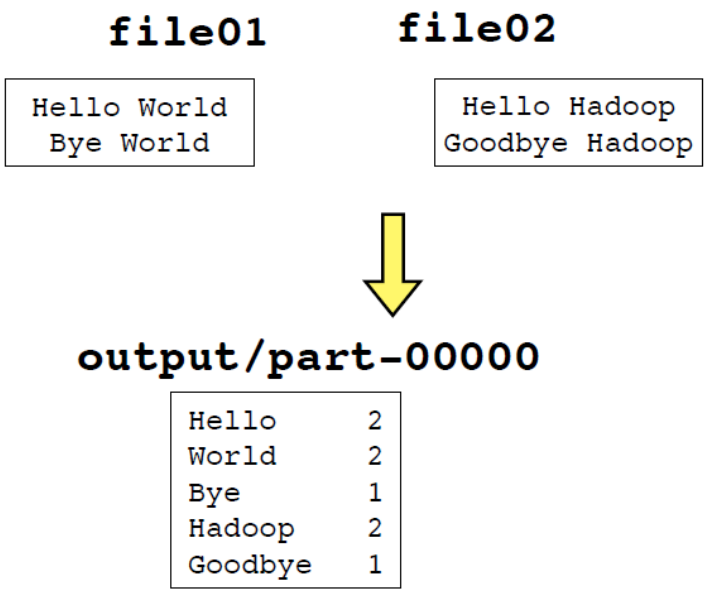


Example: Wordcount

- **How would you do this with a huge document?**
  - ↪ Each time you see a word:
    - ✓ if it is a new word, add a tick mark beside it,
    - ✓ otherwise add a new word with a tick
- **. . . But hard to parallelize**
  - ↪ pb when updating the list

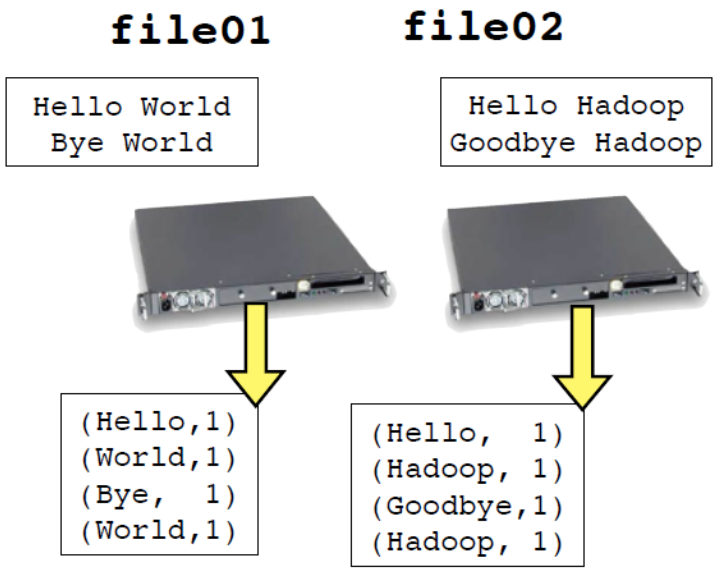


Example: Wordcount



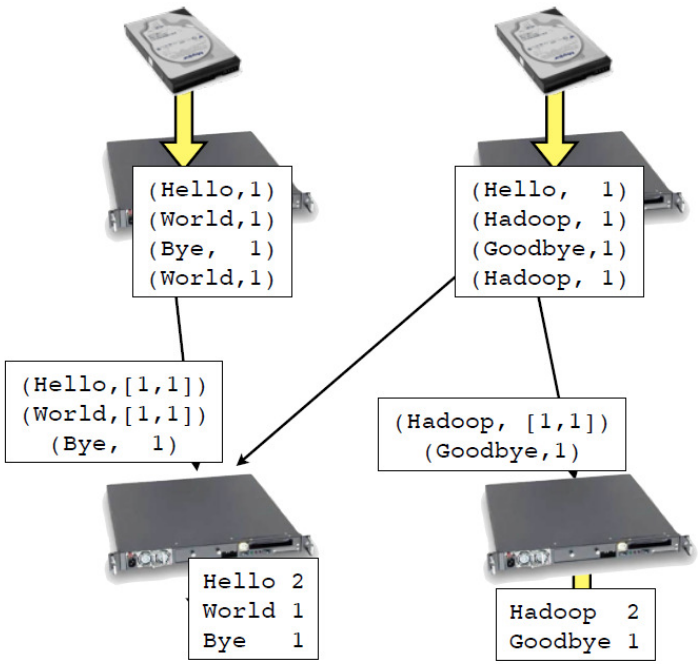
- **MapReduce way**
  - ↪ all hard work done automatically by shuffle
- **Map**:
  - ↪ just emit a 1 for each word you see
- **Shuffle**:
  - ↪ assigns keys (words) to each reducer,
  - ↪ sends (k,v) pairs to appropriate reducer
- **Reducer**
  - ↪ just has to sum up the ones

Example: Wordcount



- **MapReduce way**
  - ↪ all hard work done automatically by shuffle
- **Map**:
  - ↪ just emit a 1 for each word you see
- **Shuffle**:
  - ↪ assigns keys (words) to each reducer,
  - ↪ sends (k,v) pairs to appropriate reducer
- **Reducer**
  - ↪ just has to sum up the ones

Example: Wordcount



- **MapReduce way**
  - ↪ all hard work done automatically by shuffle
- **Map**:
  - ↪ just emit a 1 for each word you see
- **Shuffle**:
  - ↪ assigns keys (words) to each reducer,
  - ↪ sends (k,v) pairs to appropriate reducer
- **Reducer**
  - ↪ just has to sum up the ones

Hands-on 3: Playing with Hadoop

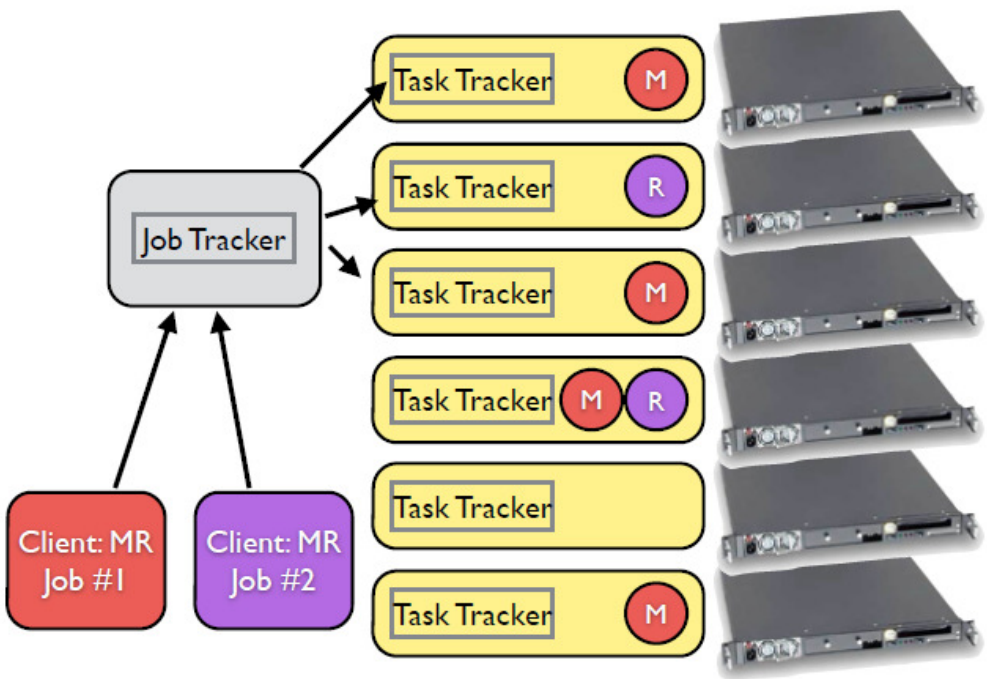
Hands-on 3

ulhpc-tutorials.rtfd.io/en/latest/bigdata/#3-running-hadoop

- Test the `tools/Hadoop` modules in Single mode — Step 3.a
  - ↪ test on a Map-reduce `grep` application
- Pseudo-Distributed Operation — Step 3.b
  - ↪ application on the official Wordcount instructions
- Enable a Full Cluster Setup — Step 3.c

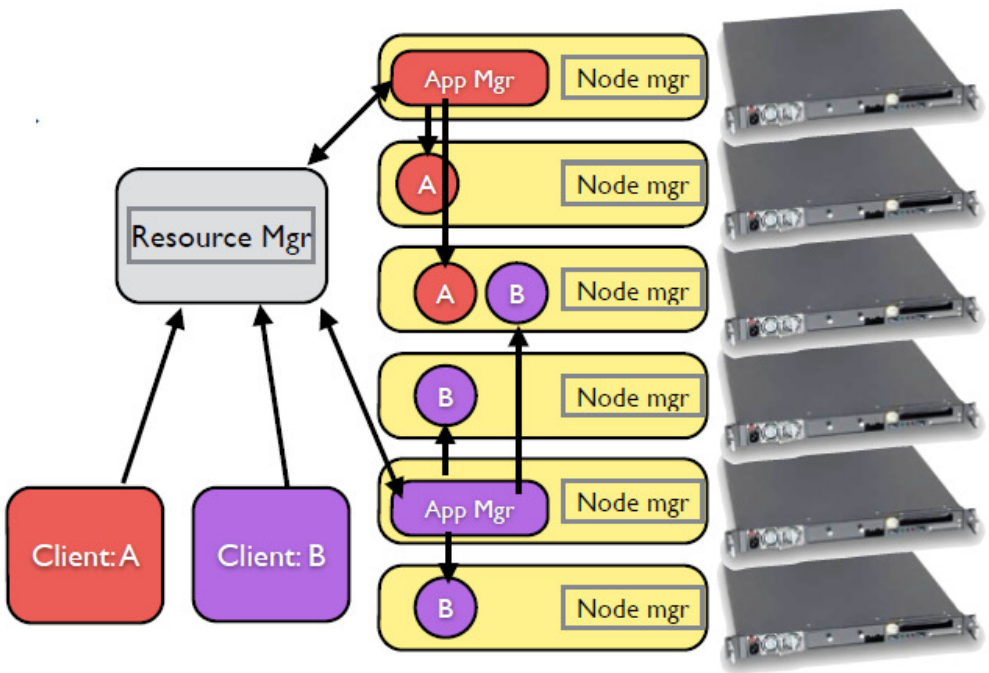
Hadoop 0.1x

- Original Hadoop was basically HDFS + infra. for MapReduce
  - ↪ Very faithful implementation of Google MapReduce paper.
  - ↪ Job tracking, orchestration all very tied to M/R model
- Made it difficult to run other sorts of jobs



YARN and Hadoop 2

- **YARN**: Yet Another Resource Negotiator
  - ↪ Looks a lot more like a cluster scheduler/resource manager
  - ↪ Allows arbitrary jobs.
- Allow for new compute/data tools. **Ex**: streaming with Spark



Apache Spark

- Spark is (yet) a(-nother) distributed, **Big Data** processing platform.
  - ↪ Everything you can do in Hadoop, you can also do in Spark.

In contrast to Hadoop

- Spark computation paradigm is not **just** MapReduce job
- Key feature - **in-memory analyses**.
  - ↪ **multi-stage, in-memory dataflow graph based on Resilient Distributed Datasets (RDDs)**.

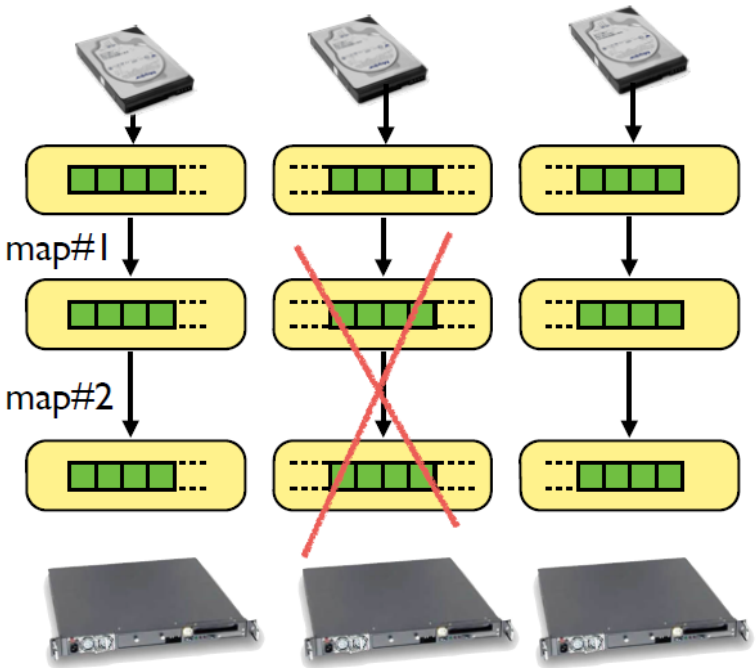
Apache Spark

- Spark is implemented in Scala, running in a Java Virtual Machine.
  - ↪ Spark supports different languages for application development:
    - ✓ Java, Scala, Python, R, and SQL.
- Originally developed in AMPLab (UC Berkeley) from 2009,
  - ↪ donated to the Apache Software Foundation in 2013,
  - ↪ top-level project as of 2014.
- **Latest release**: 2.2.1 (Dec. 2017)

RDD

- Resilient Distributed Dataset (RDD)
  - ↪ Partitioned collections (lists, maps..) across nodes
  - ↪ Set of well-defined operations (incl map, reduce) defined on these RDDs.

RDD



- Fault tolerance works three ways:
  - ↪ Storing, reconstructing lineage
  - ↪ Replication (optional)
  - ↪ Persistance to disk (optional)

RDD Lineage

- Map Reduce implemented FT by outputting everything to disk always.
  - ↪ Effective but extremely costly.
  - ↪ **How to maintain fault tolerance without sacrificing in-memory performance?**
    - ✓ for truly large-scale analyses

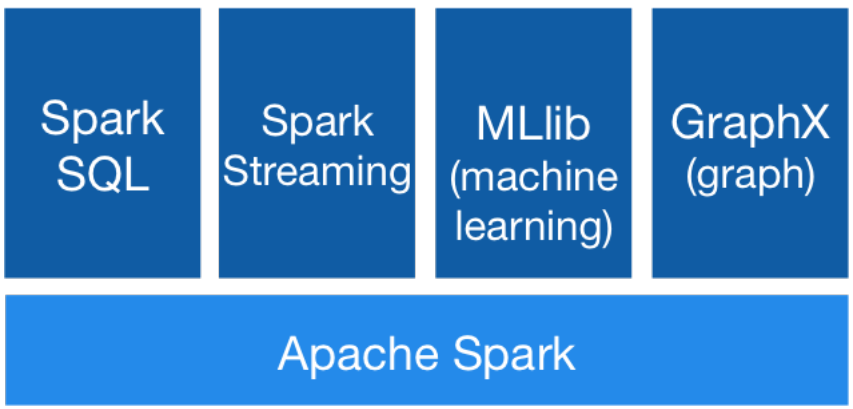
RDD Lineage

- Map Reduce implemented FT by outputting everything to disk always.
  - ↪ Effective but extremely costly.
  - ↪ **How to maintain fault tolerance without sacrificing in-memory performance?**
    - ✓ for truly large-scale analyses

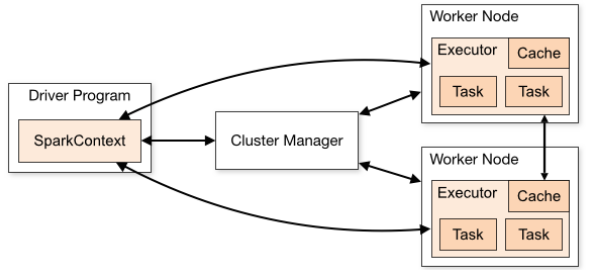
- **Solution**:
  - ↪ Record lineage of an RDD (think version control)
  - ↪ If container, node goes down, reconstruct RDD from scratch
    - ✓ Either from beginning,
    - ✓ or from (occasional) checkpoints which user has some control over.
  - ↪ User can suggest caching current state of RDD in memory,
    - ✓ or persisting it to disk, or both.
  - ↪ You can also save RDD to disk, or replicate partitions across nodes for other forms of fault tolerance.

Main Building Blocks

- The **Spark Core API** provides the general execution layer on top of which all other functionality is built upon.
- Four higher-level components (in the _Spark ecosystem):
  1. **Spark SQL** (formerly **Shark**),
  2. **Streaming**, to build scalable fault-tolerant streaming applications.
  3. **MLlib** for machine learning
  4. GraphX, the API for graphs and graph-parallel computation



Hands-on 4: Spark



Hands-on 4

ulhpc-tutorials.rtfd.io/en/latest/bigdata/#4-interactive-big-data-analytics-with-spark

- **Build Spark 2.2.0** — Step 4.1
- Check a single **interactive run** — Step 4.2
  - ↪ **PySpark**, the Spark Python API
  - ↪ **Scala Spark Shell**
  - ↪ R Spark Shell **will not be reviewed** here
- Running **Spark standalone cluster** — Step 4.3
  - ↪ launch **master** and **worker** Spark processes
  - ↪ access the **web UI** of the master
    - ✓ **Ex**: SOCKS 5 proxy approach + FoxyProxy plugin
  - ↪ submit a **sample job** (Pi estimation)
  - ↪ prepare a **launcher script**

Questions?

http://hpc.uni.lu

High Performance Computing @ uni.lu

Prof. Pascal Bouvry
Dr. Sebastien Varrette
Valentin Plugaru
Sarah Peter
Hyacinthe Cartiaux
Clement Parisot

University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
mail: hpc@uni.lu