



Large Scale File Systems

Amir H. Payberah
payberah@kth.se
2021-09-01



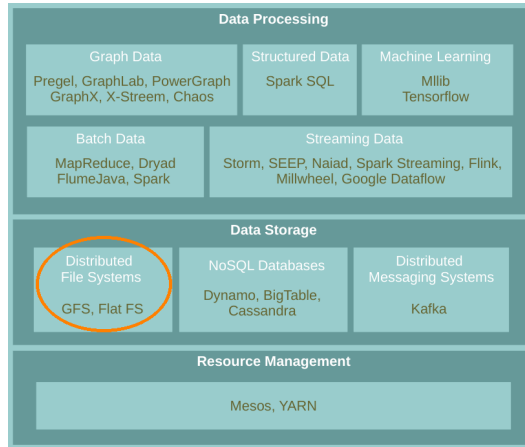


The Course Web Page

<https://id2221kth.github.io>

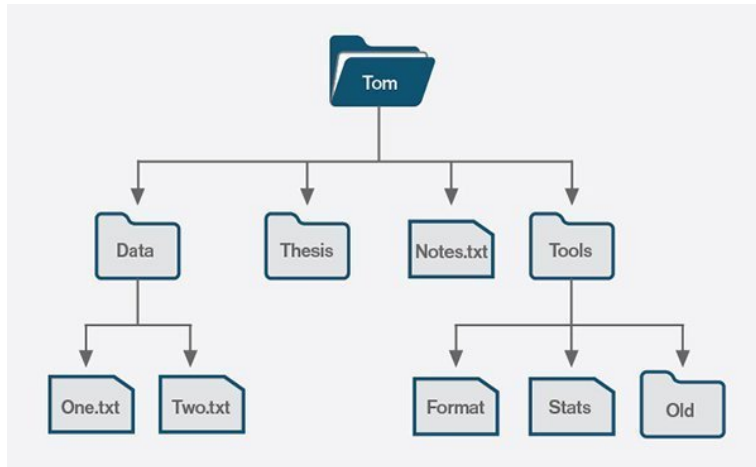
<https://tinyurl.com/f6x544h>

Where Are We?



File System

What is a File System?



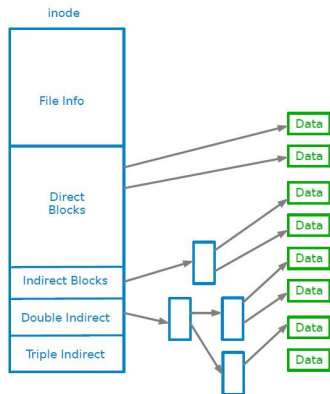
What is a File System?

- ▶ Controls how data is **stored** in and **retrieved** from **storage device**.



What is a File System?

- Controls how data is **stored** in and **retrieved** from **storage device**.





Distributed File Systems

- ▶ When data **outgrows** the storage capacity of a **single** machine: **partition** it across a **number of separate** machines.



Distributed File Systems

- ▶ When data **outgrows** the storage capacity of a **single** machine: **partition** it across a **number of separate** machines.
- ▶ **Distributed file systems**: manage the storage across a **network of machines**.

Google File System (GFS)

- ▶ Huge files (multi-GB)



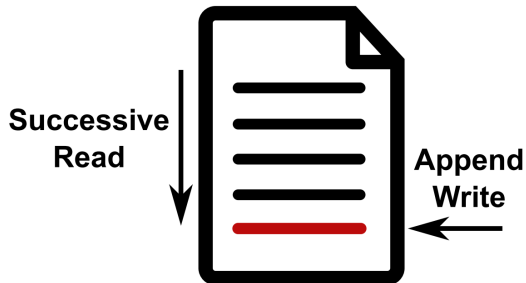
- [illegible]

- [illegible]

- [illegible]

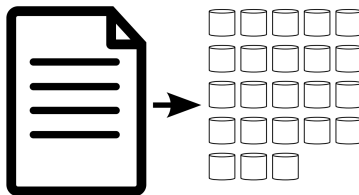
Optimised for Streaming

- Write once, read many.

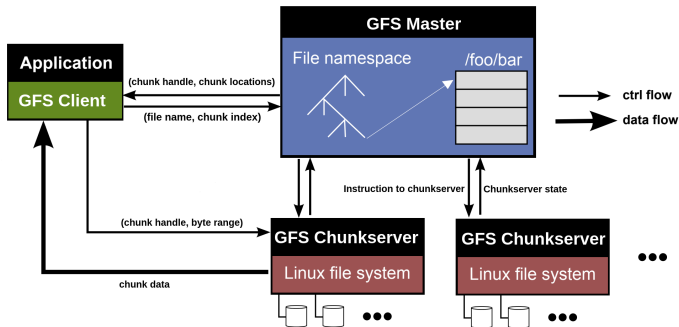


Files and Chunks

- ▶ Files are split into **chunks**.
- ▶ **Chunk**: single **unit** of storage.
 - **Immutable** and globally unique **chunk handle**
 - **Transparent** to user
 - Each **chunk** is stored as a **plain Linux file**



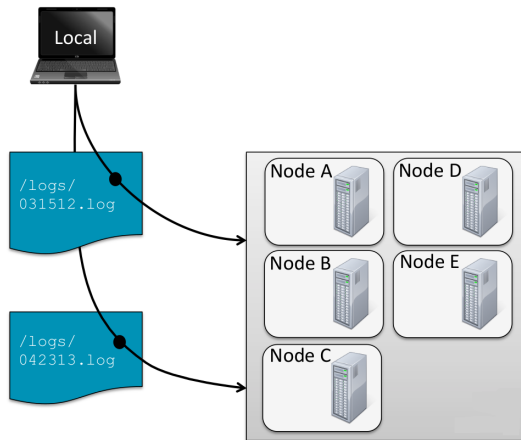
GFS Architecture



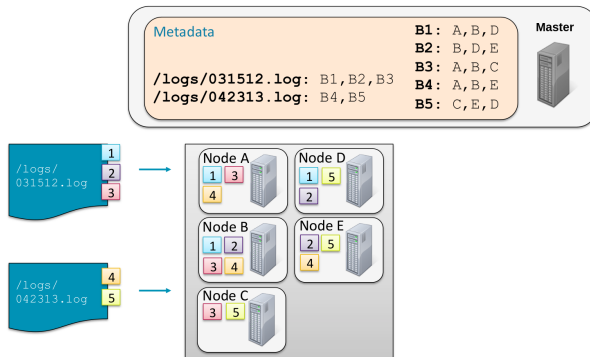
► Main components:

- GFS master
- GFS chunkserver
- GFS client

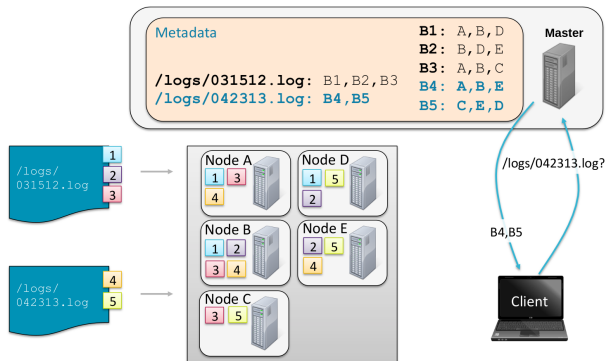
Big Picture - Storing and Retrieving Files (1/4)



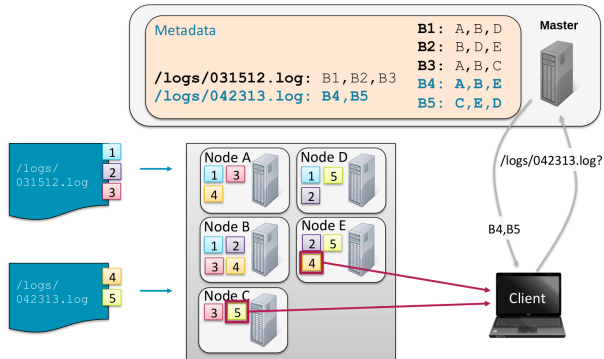
Big Picture - Storing and Retrieving Files (2/4)



Big Picture - Storing and Retrieving Files (3/4)

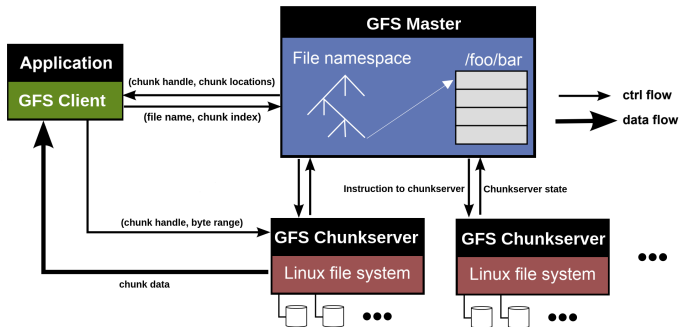


Big Picture - Storing and Retrieving Files (4/4)



System Architecture Details

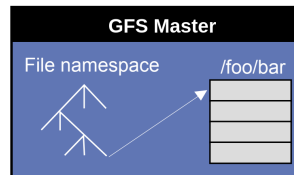
GFS Architecture





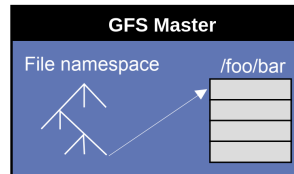
GFS Master

- ▶ Responsible for all **system-wide activities**



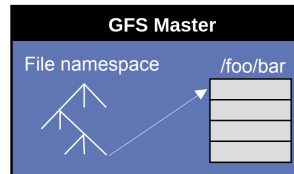
GFS Master

- ▶ Responsible for all **system-wide activities**
- ▶ Maintains all file system **metadata**
 - **Namespaces**, ACLs, mappings from files to chunks, and current locations of chunks



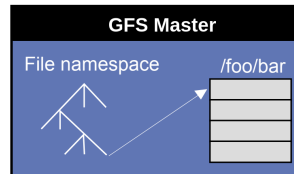
GFS Master

- ▶ Responsible for all **system-wide activities**
- ▶ Maintains all file system **metadata**
 - **Namespaces**, ACLs, mappings from files to chunks, and current locations of chunks
 - All kept in **memory**, namespaces and file-to-chunk mappings are also stored **persistently in operation log**



GFS Master

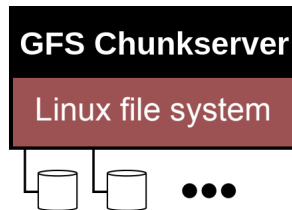
- ▶ Responsible for all **system-wide activities**
- ▶ Maintains all file system **metadata**
 - **Namespaces**, ACLs, mappings from files to chunks, and current locations of chunks
 - All kept in **memory**, namespaces and file-to-chunk mappings are also stored **persistently in operation log**
- ▶ **Periodically** communicates with each **chunkserver**
 - Determines **chunk locations**
 - Assesses **state of the overall system**





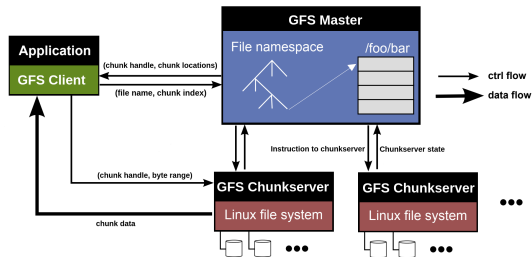
GFS Chunkserver

- ▶ Manages chunks
- ▶ Tells master **what chunks** it has
- ▶ Stores **chunks** as files
- ▶ Maintains **data consistency** of chunks



GFS Client

- ▶ Issues **control requests** to **master server**.
- ▶ Issues **data requests** directly to **chunkserver**.
- ▶ **Caches metadata**.
- ▶ Does **not cache data**.

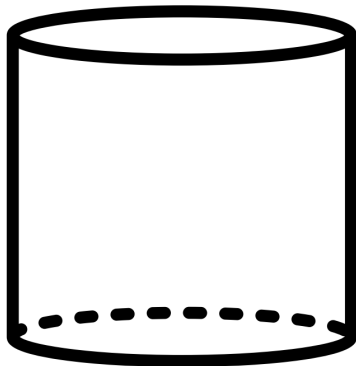




Data Flow and Control Flow

- ▶ Data flow is decoupled from control flow
- ▶ Clients interact with the master for metadata operations (control flow)
- ▶ Clients interact directly with chunkservers for all files operations (data flow)

Why Large Chunks?





Why Large Chunks?

- ▶ 64MB or 128MB (much larger than most file systems)
- ▶ Advantages
- ▶ Disadvantages



Why Large Chunks?

- ▶ 64MB or 128MB (much larger than most file systems)
- ▶ Advantages
 - Reduces the size of the metadata stored in master
 - Reduces clients' need to interact with master
- ▶ Disadvantages



Why Large Chunks?

- ▶ 64MB or 128MB (much larger than most file systems)
- ▶ Advantages
 - Reduces the size of the metadata stored in master
 - Reduces clients' need to interact with master
- ▶ Disadvantages
 - Wasted space due to internal fragmentation

System Interactions

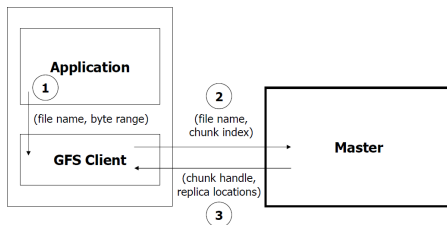


The System Interface

- ▶ Not POSIX-compliant, but supports typical file system operations
 - create, delete, open, close, read, and write
- ▶ snapshot: creates a copy of a file or a directory tree at low cost
- ▶ append: allow multiple clients to append data to the same file concurrently

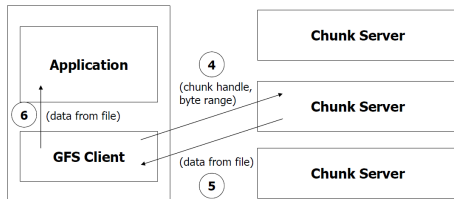
Read Operation (1/2)

- ▶ 1. **Application** originates the **read request**.
- ▶ 2. **GFS client translates** request and sends it to the **master**.
- ▶ 3. The master responds with **chunk handle** and **replica locations**.



Read Operation (2/2)

- ▶ 4. The **client** picks a **location** and sends the **request**.
- ▶ 5. The **chunkserver** sends **requested data** to the client.
- ▶ 6. The client forwards the data to the application.





Update Order (1/2)

- **Update (mutation)**: an operation that **changes** the **content** or **metadata** of a chunk.



Update Order (1/2)

- ▶ **Update (mutation)**: an operation that **changes** the **content** or **metadata** of a chunk.
- ▶ For **consistency**, updates to each chunk must be **ordered** in the same way at the **different chunk replicas**.
- ▶ **Consistency** means that replicas will end up with the **same version of the data** and not diverge.



Update Order (2/2)

- ▶ For this reason, for each chunk, one replica is designated as the **primary**.
- ▶ The other replicas are designated as **secondaries**.
- ▶ **Primary** defines the **update order**.
- ▶ All secondaries **follow** this order.



Primary Leases (1/2)

- ▶ For correctness there needs to be **one single primary** for **each chunk**.



Primary Leases (1/2)

- ▶ For correctness there needs to be **one single primary** for **each chunk**.
- ▶ At any time, **at most one server** is **primary** for each **chunk**.
- ▶ **Master** selects a **chunkserver** and grants it **lease** for a **chunk**.

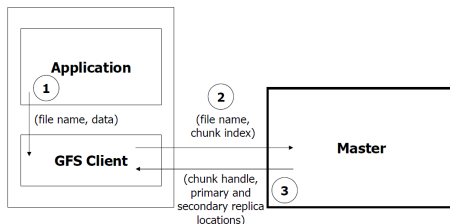


Primary Leases (2/2)

- ▶ The **chunkserver** holds the **lease** for a period T after it gets it, and behaves as **primary** during this period.
- ▶ If master does **not hear** from primary chunkserver for a period, it gives the **lease** to **someone else**.

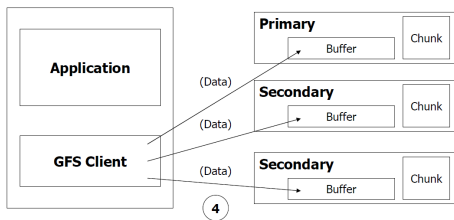
Write Operation (1/3)

- ▶ 1. **Application** originates the **request**.
- ▶ 2. The **GFS client** translates request and sends it to the **master**.
- ▶ 3. The master responds with **chunk handle** and **replica locations**.



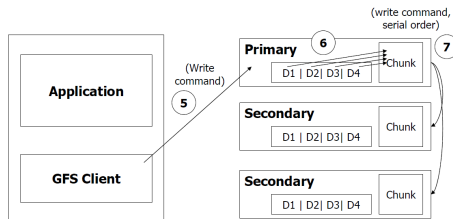
Write Operation (2/3)

- ▶ 4. The client **pushes write data** to all locations. Data is stored in chunkserver's **internal buffers**.



Write Operation (3/3)

- ▶ 5. The client sends **write command** to the **primary**.
- ▶ 6. The primary determines **serial order** for data instances in its **buffer** and writes the instances in that order to the chunk.
- ▶ 7. The primary sends the serial order to the **secondaries** and tells them to perform the write.





Write Consistency

- ▶ **Primary** enforces one **update order across** all replicas for concurrent writes.
- ▶ It also **waits until a write finishes** at the other replicas before it replies.



Write Consistency

- ▶ **Primary** enforces one **update order across** all replicas for concurrent writes.
- ▶ It also **waits until a write finishes** at the other replicas before it replies.
- ▶ Therefore:
 - We will have **identical replicas**.
 - But, file region may end up containing mingled fragments from different clients: e.g., writes to different chunks may be ordered differently by their different primary chunkservers
 - Thus, **writes** are **consistent** but undefined state in GFS.



Append Operation (1/2)

- ▶ 1. **Application** originates record **append request**.
- ▶ 2. The **client** translates request and sends it to the **master**.
- ▶ 3. The master responds with **chunk handle** and **replica locations**.
- ▶ 4. The **client** pushes **write data** to all locations.



Append Operation (2/2)

- ▶ 5. The **primary** checks if record **fits in specified chunk**.



Append Operation (2/2)

- ▶ 5. The **primary** checks if record **fits in specified chunk**.
- ▶ 6. If record **does not fit**, then the primary:
 - Pads the chunk,
 - Tells secondaries to do the same,
 - And informs the client.
 - The client then retries the append with the next chunk.

Append Operation (2/2)

- ▶ 5. The **primary** checks if record **fits in specified chunk**.
- ▶ 6. If record **does not fit**, then the primary:
 - Pads the chunk,
 - Tells secondaries to do the same,
 - And informs the client.
 - The client then retries the append with the next chunk.
- ▶ 7. If **record fits**, then the primary:
 - Appends the record,
 - Tells secondaries to do the same,
 - Receives responses from secondaries,
 - And sends final response to the client



Delete Operation

- ▶ Metadata operation.
- ▶ Renames file to **special name**.
- ▶ After certain time, deletes the actual chunks.
- ▶ Supports undelete for **limited time**.
- ▶ Actual **lazy garbage collection**.

The Master Operations



A Single Master

- ▶ The master has a **global knowledge** of the whole system
- ▶ It **simplifies** the design
- ▶ The master is (hopefully) **never the bottleneck**
 - Clients **never read and write file data** through the **master**
 - Client only requests from master **which chunkservers** to talk to
 - Further reads of the same chunk do **not involve the master**



The Master Operations

- ▶ Namespace management and locking
- ▶ Replica placement
- ▶ Creating, re-replicating and re-balancing replicas
- ▶ Garbage collection
- ▶ Stale replica detection



Namespace Management and Locking (1/2)

- Represents its namespace as a **lookup table** mapping **pathnames** to **metadata**.



Namespace Management and Locking (1/2)

- ▶ Represents its namespace as a **lookup table** mapping **pathnames** to **metadata**.
- ▶ Each master operation acquires a set of **locks** before it runs.
- ▶ **Read lock** on **internal** nodes, and **read/write** lock on the **leaf**.



Namespace Management and Locking (1/2)

- ▶ Represents its namespace as a **lookup table** mapping **pathnames** to **metadata**.
- ▶ Each master operation acquires a set of **locks** before it runs.
- ▶ **Read lock** on **internal** nodes, and **read/write** lock on the **leaf**.
- ▶ Example: **creating multiple files** (**f1** and **f2**) in the same directory (**/home/user/**).
 - Each operation acquires a **read lock** on the directory name **/home/user/**
 - Each operation acquires a **write lock** on the file name **f1** and **f2**

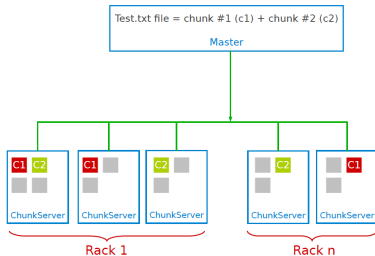


Namespace Management and Locking (2/2)

- ▶ **Read lock** on directory (e.g., `/home/user/`) prevents its deletion, renaming or snapshot
- ▶ Allows **concurrent mutations** in the same directory

Replica Placement

- ▶ Maximize data **reliability**, **availability** and **bandwidth utilization**.
- ▶ Replicas spread across machines and racks, for example:
 - 1st replica on the **local rack**.
 - 2nd replica on the **local rack but different machine**.
 - 3rd replica on a **different rack**.
- ▶ The **master** determines replica placement.





Creation, Re-replication and Re-balancing

► Creation

- Place new replicas on chunkservers with [below-average disk usage](#).
- [Limit](#) number of recent creations on each chunkserver.



Creation, Re-replication and Re-balancing

► Creation

- Place new replicas on chunkservers with **below-average disk usage**.
- **Limit** number of recent creations on each chunkserver.

► Re-replication

- When number of available replicas falls **below** a user-specified goal.



Creation, Re-replication and Re-balancing

► Creation

- Place new replicas on chunkservers with **below-average disk usage**.
- **Limit** number of recent creations on each chunkserver.

► Re-replication

- When number of available replicas falls **below** a user-specified goal.

► Rebalancing

- **Periodically**, for better **disk utilization** and **load balancing**.
- Distribution of replicas is analyzed.



Garbage Collection

- ▶ File **deletion** **logged** by master.
- ▶ File renamed to a **hidden** name with deletion timestamp.



Garbage Collection

- ▶ File **deletion** **logged** by master.
- ▶ File renamed to a **hidden** name with deletion timestamp.
- ▶ Master regularly **removes** hidden files older than 3 days (configurable).
- ▶ Until then, hidden files **can be read and undeleted**.



Garbage Collection

- ▶ File **deletion** **logged** by master.
- ▶ File renamed to a **hidden** name with deletion timestamp.
- ▶ Master regularly **removes** hidden files older than 3 days (configurable).
- ▶ Until then, hidden files **can be read and undeleted**.
- ▶ When a hidden file is removed, its **in-memory metadata is erased**.



Stale Replica Detection

- ▶ **Chunk replicas** may become **stale**: if a chunkserver fails and misses mutations to the chunk while it is down.



Stale Replica Detection

- ▶ **Chunk replicas** may become **stale**: if a chunkserver fails and misses mutations to the chunk while it is down.
- ▶ Need to distinguish between **up-to-date** and **stale replicas**.



Stale Replica Detection

- ▶ **Chunk replicas** may become **stale**: if a chunkserver fails and misses mutations to the chunk while it is down.
- ▶ Need to distinguish between **up-to-date** and **stale replicas**.
- ▶ **Chunk version number**:
 - **Increased** when master grants new lease on the chunk.
 - Not increased if replica is unavailable.



Stale Replica Detection

- ▶ **Chunk replicas** may become **stale**: if a chunkserver fails and misses mutations to the chunk while it is down.
- ▶ Need to distinguish between **up-to-date** and **stale replicas**.
- ▶ Chunk **version number**:
 - **Increased** when master grants new lease on the chunk.
 - Not increased if replica is unavailable.
- ▶ Stale replicas deleted by master in regular **garbage collection**.

Fault Tolerance



Fault Tolerance for Chunks

- ▶ Chunks replication (re-replication and re-balancing)
- ▶ Data integrity
 - Checksum for each chunk divided into 64KB blocks.
 - Checksum is checked every time an application reads the data.



Fault Tolerance for Chunkserver

- ▶ All chunks are **versioned**.
- ▶ Version number **updated** when a **new lease** is granted.
- ▶ Chunks with **old versions** are not served and are **deleted**.



Fault Tolerance for Master

- ▶ Master state replicated for reliability on multiple machines.
- ▶ When master fails:
 - It can restart almost instantly.
 - A new master process is started elsewhere.
- ▶ Shadow (not mirror) master provides only read-only access to file system when primary master is down.

GFS and HDFS

GFS vs. HDFS

GFS	HDFS
Master	Namenode
Chunkserver	DataNode
Operation Log	Journal, Edit Log
Chunk	Block
Random file writes possible	Only append is possible
Multiple write/reader model	Single write/multiple reader model
Default chunk size: 64MB	Default chunk size: 128MB



HDFS Example (1/2)

```
# Create a new directory /kth on HDFS  
hdfs dfs -mkdir /kth
```



HDFS Example (1/2)

```
# Create a new directory /kth on HDFS  
hdfs dfs -mkdir /kth
```

```
# Create a file, call it big, on your local filesystem and  
# upload it to HDFS under /kth  
hdfs dfs -put big /kth
```




HDFS Example (1/2)

```
# Create a new directory /kth on HDFS  
hdfs dfs -mkdir /kth
```

```
# Create a file, call it big, on your local filesystem and  
# upload it to HDFS under /kth  
hdfs dfs -put big /kth
```

```
# View the content of /kth directory  
hdfs dfs -ls /kth
```



HDFS Example (1/2)

```
# Create a new directory /kth on HDFS  
hdfs dfs -mkdir /kth
```

```
# Create a file, call it big, on your local filesystem and  
# upload it to HDFS under /kth  
hdfs dfs -put big /kth
```

```
# View the content of /kth directory  
hdfs dfs -ls /kth
```

```
# Determine the size of big on HDFS  
hdfs dfs -du -h /kth/big
```



HDFS Example (1/2)

```
# Create a new directory /kth on HDFS  
hdfs dfs -mkdir /kth
```

```
# Create a file, call it big, on your local filesystem and  
# upload it to HDFS under /kth  
hdfs dfs -put big /kth
```

```
# View the content of /kth directory  
hdfs dfs -ls /kth
```

```
# Determine the size of big on HDFS  
hdfs dfs -du -h /kth/big
```

```
# Print the first 5 lines to screen from big on HDFS  
hdfs dfs -cat /kth/big | head -n 5
```



HDFS Example (2/2)

```
# Copy big to /big_hdfscopy on HDFS  
hdfs dfs -cp /kth/big /kth/big_hdfscopy
```



HDFS Example (2/2)

```
# Copy big to /big_hdfscopy on HDFS  
hdfs dfs -cp /kth/big /kth/big_hdfscopy
```

```
# Copy big back to local filesystem and name it big_localcopy  
hdfs dfs -get /kth/big big_localcopy
```



HDFS Example (2/2)

```
# Copy big to /big_hdfscopy on HDFS  
hdfs dfs -cp /kth/big /kth/big_hdfscopy
```

```
# Copy big back to local filesystem and name it big_localcopy  
hdfs dfs -get /kth/big big_localcopy
```

```
# Check the entire HDFS filesystem for problems  
hdfs fsck /
```



HDFS Example (2/2)

```
# Copy big to /big_hdfscopy on HDFS  
hdfs dfs -cp /kth/big /kth/big_hdfscopy
```

```
# Copy big back to local filesystem and name it big_localcopy  
hdfs dfs -get /kth/big big_localcopy
```

```
# Check the entire HDFS filesystem for problems  
hdfs fsck /
```

```
# Delete big from HDFS  
hdfs dfs -rm /kth/big
```



HDFS Example (2/2)

```
# Copy big to /big_hdfsbackup on HDFS  
hdfs dfs -cp /kth/big /kth/big_hdfsbackup
```

```
# Copy big back to local filesystem and name it big_localcopy  
hdfs dfs -get /kth/big big_localcopy
```

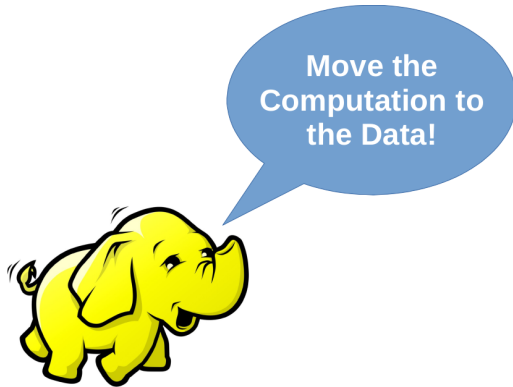
```
# Check the entire HDFS filesystem for problems  
hdfs fsck /
```

```
# Delete big from HDFS  
hdfs dfs -rm /kth/big
```

```
# Delete /kth directory from HDFS  
hdfs dfs -rm -r /kth
```


Flat Datacenter Storage (FDS)

Motivation and Assumptions (1/5)



- ▶ Why **move computation close** to **data**?
 - Because **remote** access is **slow** due to **oversubscription**.

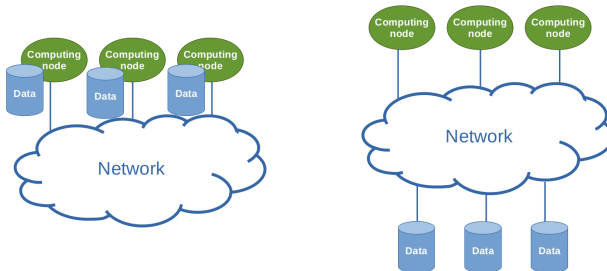


Motivation and Assumptions (2/5)

- ▶ Locality adds complexity.
- ▶ Need to be aware of where the data is.
 - Non-trivial scheduling algorithm.
 - Moving computations around is not easy.

Motivation and Assumptions (3/5)

- ▶ Datacenter networks are getting **faster**.
- ▶ Consequences
 - Support full **bisection bandwidth**: no **local** vs. **remote** disk distinction.
 - Simpler work **schedulers** and **programming models**.





Motivation and Assumptions (4/5)

- ▶ File systems like GFS manage metadata **centrally**.
- ▶ On every **read** or **write**, clients contact the **master** to get information about the location of blocks in the system.



Motivation and Assumptions (4/5)

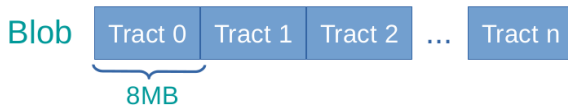
- ▶ File systems like GFS manage metadata **centrally**.
- ▶ On every **read** or **write**, clients contact the **master** to get information about the location of blocks in the system.
 - Good **visibility** and **control**.
 - **Bottleneck**: use **large** block size
 - This makes it **harder** to do **fine-grained** load balancing like our ideal little-data computer does.

Motivation and Assumptions (5/5)

- ▶ Let's make a **digital socialism**
- ▶ **Flat** Datacenter Storage



Blobs and Tracts



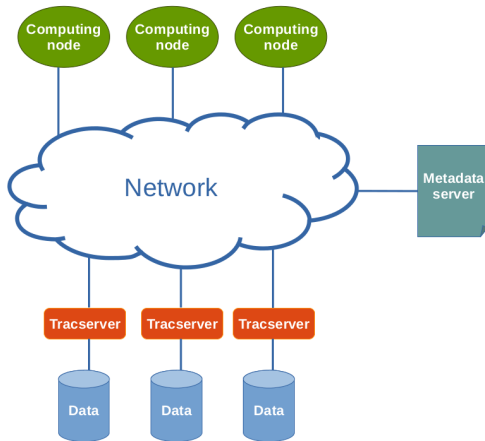
- ▶ Data is stored in logical **blobs**.
 - Byte sequences with 128-bit Global Unique Identifiers (**GUID**).
- ▶ Blobs are divided into **constant sized** units called **tracts**.
 - Tracts are sized, so **random** and **sequential** accesses have same throughput.
- ▶ Both tracts and blobs are **mutable**.



FDS API

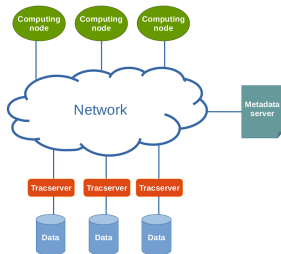
- ▶ Reads and writes are atomic.
- ▶ Reads and writes not guaranteed to appear in the order they are issued.
- ▶ API is non-blocking.
 - Helps the performance: many requests can be issued in parallel, and FDS can pipeline disk reads with network transfers.

FDS Architecture



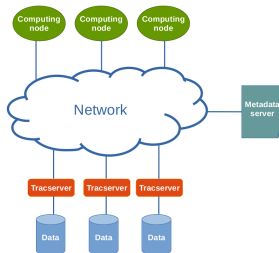
Tractserver

- ▶ Every **disk** is **managed** by a process called a **tractserver**.
- ▶ Tractservers accept commands from the network, e.g., **ReadTract** and **WriteTract**.
- ▶ They do **not use file systems**.
 - They lay out **tracts** directly to disk by using the **raw disk** interface.



Metadata Server

- ▶ Metadata server coordinates the cluster.
- ▶ It collects a list of active tractservers and distribute it to clients.
- ▶ This list is called the tract locator table (TLT).
- ▶ Clients can retrieve the TLT from the metadata server once, then never contact the metadata server again.



Track Locator Table (1/2)

- ▶ **TLT** contains the address of the **tractserver(s)** responsible for tracts.
- ▶ Clients use the blob's GUID (**g**) and the tract number (**i**) to select an **entry** in the TLT: **tract locator**

$$\text{TractLocator} = (\text{Hash}(g) + i) \bmod \text{TLT Length}$$

Locator	Disk 1	Disk 2	Disk 3
0	A	B	C
1	A	D	F
2	A	C	G
3	D	E	G
4	B	C	F
...
1,526	LM	TH	JE



Track Locator Table (2/2)

- ▶ The only time the TLT changes is when a **disk fails** or is **added**.
- ▶ **Reads and writes** **do not** change the TLT.
- ▶ In a system with more than one replica, **reads** go to **one** replica at random, and **writes** go to **all of them**.



Per-Blob Metadata

- ▶ **Per-blob metadata**: blob's length and permission bits.
- ▶ Stored in **tract -1** of each **blob**.
- ▶ The **tractserver** is responsible for the blob **metadata tract**.
- ▶ Newly created blobs have a length of **zero**, and applications must **extend** a blob before writing. The extend operation is **atomic**.

Fault Tolerance



Replication

- ▶ **Replicate** data to improve **durability** and **availability**.
- ▶ When a disk **fails**, redundant copies of the **lost data** are used to restore the data to full replication.

Replication

- ▶ **Replicate** data to improve **durability** and **availability**.
- ▶ When a disk **fails**, redundant copies of the **lost data** are used to restore the data to full replication.
- ▶ **Writes a tract**: the client sends the write to **every tractserver** it contains.
 - Applications are notified that their writes have **completed** only after the client library receives **write ack** from **all replicas**.
- ▶ **Reads a tract**: the client selects a **single tractserver** at random.

Failure Recovery (1/2)

- ▶ **Step 1:** Tractservers send heartbeat messages to the metadata server. When the metadata server detects a tractserver timeout, it declares the tractserver dead.
- ▶ **Step 2:** invalidates the current TLT by incrementing the version number of each row in which the failed tractserver appears.
- ▶ **Step 3:** picks random tractservers to fill in the empty spaces in the TLT where the dead tractserver appeared.

Row	Version	Replica 1	Replica 2	Replica 3
1	8	A	F	B
2	17	B	C	L
3	324	E	D	G
4	3	T	A	H
5	456	F	B	G
6	723	G	E	B
7	235	D	V	C
8	312	H	E	F

Row	Version	Replica 1	Replica 2	Replica 3
1	9	A	F	H
2	18	T	C	L
3	324	E	D	G
4	3	T	A	H
5	457	F	C	G
6	724	G	E	A
7	235	D	V	C
8	312	H	E	F

Failure Recovery (2/2)

- **Step 4:** sends **updated TLT assignments** to every **server** affected by the changes.
- **Step 5:** waits for each **tractserver** to **ack** the new TLT assignments, and then begins to give out the new TLT to clients when queried for it.

Row	Version	Replica 1	Replica 2	Replica 3
1	8	A	F	B
2	17	B	C	L
3	324	E	D	G
4	3	T	A	H
5	456	F	B	G
6	723	G	E	B
7	235	D	V	C
8	312	H	E	F

Row	Version	Replica 1	Replica 2	Replica 3
1	9	A	F	H
2	18	L	C	L
3	324	E	D	G
4	3	T	A	H
5	457	F	C	G
6	724	G	E	A
7	235	D	V	C
8	312	H	E	F

Summary



Summary

- ▶ Google File System (GFS)
- ▶ Files and chunks
- ▶ GFS architecture: master, chunk servers, client
- ▶ GFS interactions: read and update (write and update record)
- ▶ Master operations: metadata management, replica placement and garbage collection



Summary

- ▶ Flat Datacenter Storage (FDS)
- ▶ Blobs and tracts
- ▶ FDS architecture: Metadata server, tractservers, TLT
- ▶ FDS interactions: using GUID and track number
- ▶ Replication and failure recovery



References

- ▶ S. Ghemawat et al., The Google file system, Vol. 37. No. 5. ACM, 2003.
- ▶ E. Nightingale et al., Flat Datacenter Storage, OSDI, 2012.

Questions?