Google File System. Revolutionary. No Legacy Issues. Non POSIX API. Only Performance Matters. Strongly fault tolerant. Automatic Recovery. Multi GB Files, 10*9 Objects.
Most modifications: Appends. Almost no Random Writes. WORSequentially.
Reads: Large Streaming / Random Small forward reads.
Designed for Sustained BW >> Latency. Special Operations: Snapshots / Record Appends.

Master: No Dir. No Hard/Symb links. Full path to MD mapping (with Prefix Kompr)
Single Master, Why? Simple. No Synch. Easier Writes. Only store chunk locations. Not data. Batch requests for Chunks. Optimistic Prefetching of Chunk Handles not requested for.
Chunks: 64 MB(!!). Small MD NS & lesser entries (In Mem MD). Fewer Chunk Locations on request. (-) Fragmentation.
MD: (File & Chunk NS. File Chunk Mapping. Location of 3)
In memory (because fast > fast global scans(GC & reorg) > 64B MD / 64MB data. Prefix Compression (abcd.txt->7XN).
Chunk Locations: Poll at startup. Use heartbeat to monitor. Not on demand (better when failures are infrequent)
Logging & recovery: MD: <old, new>. Log replicated on R. Global Snapshots>Trunc Logs. Recover:Last Snap+logs after
Consistency model: Relaxed. Single Master-> Concur changes consistent. +Leases help. Append is atomic 1ce. Edits are synchronous. Versioning for misses (Lamport's)
Update: 60 sec lease to 1 replica (who takes care of others) Lease may be extended. Piggybacked on heartbeat.

Snapshot Howto: COW. Revoke outst leases. Start Logging upd.Commit log.Apply log to MD.No chunk copy till nxt upd
Locks: Lock all directories to file (read lock; concur updates to directory OK). Lock file (write lock). Total ordering of locks (deadlock prevention).
GC: Better than eager deletion (as unfinished replica creation; lost deletion msgs). GC collected after 3 days. Runs alongside other operations(snapshots, locks, upd) in co-ord. Safety against accidents.
Fault Tolerance: Fast recovery (state restoration). Chunk Replication. Ghost master for read only during master restoration interim. Data Integrity(64KB blocks of chunks. Have checksum. Verified during writes/reads/background)
Reads>>Writes.Appends>>Writes.(Opt for common case)
USP: FS API-> Tailored. Single Master. In Mem MD. Flat NS.

Dedup: +less disk +less bw +less power –more compu time – more complex – latent update channels.

LBFS Insertions.

ZFS: Pooled Storage. No Volumes. Transactional Obj. Storage: Always consistent on disk. Provable End-end data integrity: No Corruption.

ZFS IO Stack: 1. ZPL (ZFS Posix Layer) Make x changes to Y objects(Atomic). 2. DMU: Data Mngm Unit. Transactional Group Commit (Atomic for group, always consistent, not J needed). 3. SPA: Storage Pool Access: Sched, Agg, Issue IO on demand. No Resync on power loss.Runs at platter speed
DMU: Trans Obj Store: 2^48 obj of 2^64 B each.
COW:(Diag) Untouched Block Tree>COW of Leaf nodes>COW indirect blocks>Rewrite Root to point to new tree(**Hence atomic write**). Bonus: Old tree->Snapshot.
ZFS Birth Time:

Prune to get correct data snapshot at any point.
Errors: RAID only picks noisy errors. ZFS: Validation with CRC for every node of tree.

Validate entire path. Traditional RAID fetches corrupt block
ZFS: Detect first copy corrupt using CRC. Try second disk. Fetch and repair corrupt disk.
Stripe regeneration: RAID4/5: Flaw:Partial Stripe writes->regenerate usingRMW (Slow!). Done for whole block, even if mostly empty. Raid Z: Dynamic Stripe Width(each logical block-> own stripe). All writes are full stripe writes. No RMW reqd. Detects silent data Corruption.
Traditional Resilvering: Copy old disk to new. Whole disk copy, even if mostly empty. No CRC midway. Root may be last to get copied !!
ZFS: Top down: most imp first. Only live (tree) blocks. Multiple Failures: RAID may not handle. Silent errors, compound issues. Stop block tree from compromise -> higher block > more replicated
Disk Scrubbing: Find and Correct latent errors. Verify <&Repair> ALL data (data, MD, parity, ditto) against 256b CRC. Low IO priority: Doesn't reduce performance.
ZFS Scalability: 128b: Quantum limit of data on earth**.**
Concurrent: Byte range locking, Pipelined, parallel R/W as much as possible. Parallel, constant time dir ops.
ZFS Performance: 1. COW (Random Write -> sequential). 2.Pipelined IO to allow max IO Parallelism. 3. Dynamic Striping -> new disk? Start striping for new data, gently reallocate old data. 4.Intelligent Prefetch: Auto length/stride detection: next row/next column fetch. 5.Variable block size: Per object granularity. Transparent block based compression.
ZFS Snapshots: Read only, point-in time copy. Instantans, unlimited in #. No additional space used (COW). Accessible thru .zfs/snapshot.
ZFS Clones: Writable copy of Snapshot. Many copies of mostly shared data.
ZFS Summary: Simple, Powerful (Snapshots, pooling, clone, compression, scrubbing, RAIDZ), Safe (Error detec, correct, silent data corruption), Fast (Dynamic Striping, Intelligent Prefetching, Pipelined IO), Open Src / Free.

LBFS: Exploit similarity in files/versions of same file. Avoid sending data already with Server/ in client cache. Also use conventional compression & caching.
LBFS Server: Divide files into chunks and index by hash.
LBFS Client: Maintain large prstnt cache index for chunks Space(index) vs. Time(b/w). MD vs Data. Compress all transfers (Processing vs. NW BW). LBFS Provides Close to Open File Consistency. Client tries to reconstitute files using cached chunks instead of requesting from the server. **LBFS Chunking:** Non overlapping chunks. Chunk boundaries based on contents. Examine regions for breakpoints using Rabin Fingerpr. 8KB + 48B window.
Indexing: Not very small(hash~data). Not very large(!transfer in single RPC). 2KB < LBFS Chunk<64KB.
Chunk DB: Index using 1st 64b of SHA1 hash. Always recompute before using(Simpler crash recovery & synch, helps with has collisions).
Reading: New file? GetHash. Check which chunks not present. Request those chunks. Server sends those chunks. Reconstruct file using new+cached chunks.
Writing: New file? Break to chunks. SendHash to server. Server requests chunks it doesn't have. Send those chunks. Server says OK. Close file.
Protocol: NFSv3+GetHash to exploit commonality+Leases for consistency. Komprs using gzip.
File Consistency: Get lease on RPC of LBFS File > If lease expired, get new lease, check hashes > if changed, request new version from server. No write leases needed (C-Open consistency). Server never demands dirty file back. Concurrent writes? Last one wins. Updates are atomic. **Architecture:**

GNR: Supercomputer: Compute Bound-> IO Bound GPFS High Throughput: Wide Striping: MD and Data Striped across Many disks. Files Striped blockwise across ALL disks. **RAID Disk Failure:** 1st failure: Degraded Rebuild, low priority. 2nd Failure: Critical Rebuild, highP.
Storage Stack: 1. Application-Compu. 2. GPFS || FS – Shared Access.3.RAID– Reliability.4.Physical Disk – Space
External RAID: Traditional Rebuild severely reduces Performance. – failures increase with complexity & rebuild time with Size.. So drive rebuild becomes more and more important - Cannot handle Silent Data Corrupt
Use GPFS w/ native RAID: No external RAID Contrlr. Direct connection to Disk Array.
Native Raid: Higher Perf: Use declustered RAID to minimize performance degradation during rebuild. Extreme Data Integrity: Use end-end checksums and version # to detect, locate&correct silent data corruptn.
Declustered RAID: 3 groups – 6 disks+1 spare: 49 stripes – mix match bhel puri. Max 2 of same color in one column. Allow rebuild if one fails. **Rationale:** Rebuild spread across many disks-> faster rebuild or less disruption to user programs.
Declustered RAID6: 14 disks 3x4+2 spares (RAID 6->2). Spread across all disks. Reduce MAX faults per stripe > rebuilding is easier.
Checksums & version#: Checksums detect corruption. Only a validated checksum can protect against dropped writes. – Old MD matches old Checksums. –Version #s in metadata to validate checksums. End-End checksum: Write Op: Comput Node->IO Node-> Disk with version#. Read Op: Disk -> IO Node with version# -> comput Node

Replication:

Power: 3 States: Active, Standby, Off. Power consumption is dominated by motor. Read ~ Write power. Arm positioning is trivial. Saving Power: Turn off Disk. Modify request stream to increase idle period. **Total Energy:** Active (Seeking, Rotating, Reading, Writing), Idle (Low Power mode, Transition between modes). Seek energy = f(seek distance). {Ro, Re, Wr} is a linear function of {Ro, Re, Wr}. Power models: Dempsey {Active Periods: Seeking, Rotation, Reading, Writing; Idle Periods: Idle Period Energy Profile}. 2 Param {Active: Single fixed average power, Idle; Single fixed average power}, 3 Param {Active Periods: Single Fixed Average Power; Idle Period: Two power modes, fixed waiting threshold for transition}.
Write Offloading: Long Periods of Write only time. Increase off time by temporarily writing to online disk. Move writes later, after disk comes online. Wake up disk only in response to demand reads.

SNIA:

File/ Record Layer: Access Methods: FS/DB. Function: Pack fine grain to large grain data. Fine grain naming, space alloc. Also: Caching for perf., coherence in distributed systems.
Block Layer: Connect with storage device. Functions: Block Aggregation, address mapping, in SAN Aggregation, "Virtualization", slicing, concatenation, striping. Local and remote mirroring (RAID-n). Examples: Volume Managers, disk Array LUs. Also do caching.
Block Layer may do: Space mngmt. Strping(for performance), Redundancy(full[Mirroring]/partial[RAID 4/5/6], point in time copy)
Possible architectures: Direct Attached, SN Attached{Cloud in between}, SN Aggregation {m:n cloud}.

MultiSite Block Storage: Data exchange across: Device-device WAN, Agg. Appliance –Agg Appliance WAN, Host to Host WAN. **SAN vs. NAS:** Map by requirement: FC vs Ethernet, FAP vs BAP. **Grid:**

Replica: Copy not parity. Uses: Backup(reliability), recovery (availability), decision support, testing platform, migration, compliance. **Kinds of Replica:** Point in time(Zero RPO), continuous: non zero RPO. Remote replication: **Diagram turnover. Synchronous:** Atomic, Slower. Network must support synch thru. **Asynch:** Non, Faster, Mean throughput

WAFL: Write anywhere File Layout. **Goals:** Provide Fast NFS. Dynamic storage support. High Performance over RAID. Tolerate Disk Failure. **Snapshots:** Read only copy of entire FS. Recover from crashes/accidents. **WAFL design:** 4KB Blocks. Store MD in files. 3type of MDs: Inode File (root Inode for F), BlockMap File(Free blocks), Inode Map(Free Inodes).

MD in Files, why? Write MD anywhere. Efficiency with RAID incre. Works with RAID4. Multiple writes to same RAID Stripe. No 4-1 short write penalty (bottleneck on parity drive). Increase Size on the fly.
Snapshot: Create duplicate of root Inode. Call It snapshot Inode. Use COW to capture latest changes as required. Older tree remains untouched, and pointed to by the snapshot Inode. -> Snapshot creation is quick, updates gradually travel up the tree, writes are batched before disk write episode. 1 new tree for many batched writes.
CP: Checkpoints every 10secs. not accessible to user.
Between 2 CPs: NFS Requests are logged on NVRAM(fast!!). In use blocks aren't overwritten. Previous CP remains unchanged. State advances atomically between CPs. Crash? **Restore Latest CP + all log entries**. Why Log? Better than caching at disk level, more predictable. NVRAM much faster than disk. Possible to keep logging + Makes CP generation and saving atomic. 1000 operations per MB of NVRAM. Remove Disk writes from the write path! Write sequentially to disk when NVRAM gets full.Sequential much faster than Random
WAFL Optimized for Writes, why? Write Performance Important for NFS Servers. Read caches are large, & at both ends. => Writes become bottlenecks. Disks on an NFS Server may have upto 5times #writes as #reads.

Business Continuity: Preparation for, response to, and recovery from an application outage that adversely affects business Ops. **Solns:** Address system unavailability, or degraded application performance & recovery strategies. **Why?** Lost productivity, reputation, revenue, temp. employees for repair.

Redundancy at each level is used to address reliability: Mirrored RAID, mirrored Cloud, 2 networks, 2 switches, aux host, HBA / Port pairs.

Zero RPO: Synchronous replication. Only reason.
Full, Cumulative, incremental backups.
Backup vs. Archiving

DAFS: Types: **1. Raw block (**get & put block, hi perf., application must do data management, OS support reqd, architecture support required.**) 2. Local FS(** good perf., data sharing not supported. **3. Cluster FS (**Each node executes Block access protocols, eg. SAN**). 4. FAP (**NFS, CIFS, virtualization, fine grain DM, = data sharing with protection.**)**

Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.
Hadoop Common – contains libraries and utilities needed by other Hadoop modules;**Hadoop Distributed File System (HDFS)** – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster; **Hadoop YARN** – a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications;and **Hadoop MapReduce** – an implementation of the MapReduce programming model for large scale data processing.
MR Wordcount:

```
function map(String name, String document)
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator part
  // word: a word
  // partialCounts: a list of aggregated p
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

MapReduce functions. The hot spots, which the application defines, are: an input reader, a Map function, a partition function, a compare function, a Reduce function, an output writer.
Stateful vs Stateless: Stateless approaches are easier to build but we know much less about the state in which the system may find itself after a crash. Transactional approaches are well supported but really assume a database style of application (data lives "in" the database). Other mechanisms we'll explore often force us to implement hand-crafted solutions in our applications and depart from the platform architectural standards (or go "beyond" them). Companies hesitate when faced with such options because they are costly to support.
A stateful server remembers client data (state) from one request to the next. A stateless server keeps no state information. Using a stateless file server, the client must specify complete file names in each request specify location for reading or writing re-authenticate for each request. Using a stateful file server, the client can send less data with each request. A stateful server is simpler. On the other hand a stateless server is more robust lost connections can't leave a file in an invalid state rebooting the server does not lose state information rebooting the client does not confuse a stateless server
Eager Dedup: - Slows down write path (waste time), + save space, - Does not handle latent changes as well as Lazy, +Predictable Disk usage & empty disk space remaining, - Performance vulnerable to transient, frequently changing / volatile data, + Better when most incoming data is duplicated (eg. Archival Store.)