- eThe File locking is possible

Stateless: server maintains no information on client accesses

- [ ] Each request identifies file and offsets
- [ ] Server can crash and recover: no state to lose
- [ ] Client can crash and recover (as usual)
- [ ] No open()/close() needed as they only establish state
- [ ] No server space used for state
    - Great for scalability: gimme more clients, I don't know them, I don't care!
- [ ] But what if a file is deleted on server while client is working on it?
- [ ] File locking (and, potentially, transactions) not p it iiik kiossible

Caching

Write-through caching: every change is propagated to master copy

- What if another client reads its own (out-of-date) cached copy?
- All accesses will require checking with server
- Or, server maintains state and sends invalidations

Write-behind caching: delay the writes and batch them

- Data buffered locally (and others don't see updates!)
- Remote files updated periodically
- One bulk write is more efficient than many little writes
- Problem: ambiguous semantics

Read-ahead caching: be proactive and prefetch data

- Request chunks of file (or the entire file) before it is needed
- Minimize wait when it actually is needed

Write-on-close caching: implement session semantics and be done with it

Centralised control

- Server is responsible for keeping track of who has what open and cached on each node
- Stateful file system, excessive traffic

Andrew File System AFS

Key assumptions

- Most files are small
- Reads are more common than writes
- Most files are accessed by one user at a time
- Files are referenced in bursts and once referenced, a file is likely to be referenced again (spatial and temporal locality)

Design decisions

- Whole file serving: on open() send the entire file
    - Whole file caching
    - Client caches entire file on local disk

    - Client writes the file back to server on close()
        - If modified
        - Keeps cached copy for future accesses
- Each client has an AFS disk cache
    - Part of disk devoted to AFS (e.g., 100MB)
    - Client manages cache using LRU
- Clients communicate with set of trusted servers
- Each server presents one identical name space to clients
    - All clients access it in the same way
    - Location transparent

File management and access

- Information service: the Volume Location Server (VLS) is a directory of cells and hosts the Volume Location Database (VLDB)
    - All the nodes of the system see the same name space in the form /afs/cellname/path
    - For example, afs/inf.ed.ac.uk/home/derp/code/src/crash.c
- Read-only volumes may be replicated on multiple servers
- To access a file
    1. Traverse AFS mount point, e.g., /afs/inf.ed.ac.uk
    2. AFS client contacts VLDB on VLS to look up the volume
    3. VLDB returns volume id and list of machines (≥ 1) maintaining file replicas
    4. Request root directory from any machine in the list
    5. Root directory contains files, subdirectories, and mount points
    6. Continue parsing the file name until another mount point (from previous step 5) is encountered; go to step 2 to resolve it

Caching

- On file **open**()
    - Server sends entire file to client and provides a **callback promise** it will notify the client when any other process modifies the file (possible due to write-through caching)
- If a client modifies a file, contents are written to server on file close()
- When a server detects an update
    - Notifies all clients that have been issued the callback promise
    - Clients invalidate cached files
- If a client goes down, then it must recover
    - Contact server with timestamps of all cached files to decide whether to invalidate
- Session semantics: if a process has a file open, it continues accessing it even if it has been invalidated
    - Upon close(), contents will be propagated to server; last update wins

Pros and cons

Pros	**Cons**
Scales well Uniform name space Read-only replication Security model supports mutual authentication, data encryption (though we didn’t talk about those)	Session semantics Directory-based permissions Uniform name space

Google File System - GFS

Key assumptions

Things fail; deal with it

- Thousands of nodes
- Bugs and hardware failures are out of file system designer's control
- Monitoring, error detection, fault tolerance, automatic recovery

Files are much larger than traditional standards

- Single file size in the order of multiple gigabytes
- Billions of files constantly served

Modifications are mainly appends

- Random writes are practically nonexistent
- Many files are written once, and read sequentially

Two types of reads

- Large streaming reads
- Small random reads (but in the forward direction)

Sustained bandwidth more important than latency

Architecture

GFS Cluster

- A single **master**, with multiple **chunkservers** per master
- Each chunkserver is running a commodity Linux OS and FS

GFS file

- Represented as fix-sized chunks (64mb each)
  - thus minimising number of requests to master and overhead of chunk access
- A chunk is divided into 64kB blocks, each with its checksum
  - Verified at read and write times
- Each chunk with a 64-bit unique global ID
- Stored mirrored across chunkservers for fault tolerance

Master server

- maintains all the **metadata** in its memory
  - 64 bytes of metadata per 64MB of data
  - File and chunk name spaces
  - File-to-chunk mappings
  - Locations of a chunk's replicas
  - Name space, access control, garbage collection, chunk migration
- flat design - No directories, no hierarchy, only a mapping from metadata to path name
- Master answers queries only about chunk locations
  - the client fetches the chunks from the chunkserver
- A client typically asks for multiple chunk locations in a single request
- The master also proactively provides chunk locations immediately following those requested (a la read-ahead, but only for metadata)

- Chunkservers are monitored through “heartbeat” messages; if a server is dead, use one of the other chunkservers to retrieve a chunk’s replica
- Master and chunkservers are designed to restore their states and start in seconds regardless of termination conditions
- **Shadow** masters provide read-only access when the primary master is down

GFS clients
- Consult master for metadata
- Access data directly from chunkservers
- No caching at clients and chunkservers due to the frequent case of streaming

Consistency model

[It would be cool if someone who understands the consistency models adds a bit of explanation to this section.]

Relaxed consistency: concurrent changes are consistent but their order is undefined (first to commit wins)
- An append is atomically committed at least once
- Then, all changes to a chunk are applied in the same order to all replicas
- Primitive versioning to detect missed updates

To update a chunk
- The master grants a chunk lease to a replica, which determines the order of updates to all replicas
- The lease has a timeout of 60s, but can be extended
- If a lease times out, the master assumes the server is dead and grants a lease to different server

Replication objectives
- Maximize data reliability and availability, and network bandwidth
- Chunk replicas are spread across physical machines and racks
- Each file has a replication factor (i.e., how many times its chunks are replicated); low replication factor → higher priority

Replication

Why?
- Enhance reliability
    - Correctness in the presence of faults or errors
    - For example, while at least one of the AFS servers has not crashed, data is available
    - Remote sites working in the presence of local failures
- Improve performance
    - Load sharing
    - Alternative locations to access data from
    - Data movement minimisation

Replication requirements

- Transparency: clients see logical objects, not physical ones, but each access returns a single result
- Consistency: all replicas are consistent for some specified consistency criterion

CAP

- **Consistency**: all nodes see the same data at the same time
- **Availability**: node failures do not prevent system operation
- **Partition tolerance**: link failures do not prevent system operation

<u>A distributed system can satisfy any two of these guarantees at the same time, but not all three</u>

Synchronisation models

No explicit synchronisation

- **Strict** - Absolute time ordering of all shared accesses matters
- **Linearisability** - All processes must see all shared accesses in the same order; accesses are ordered according to a global timestamp
- **Sequential** - All processes see all shared accesses in the same order; accesses are not ordered in time
- **Causal** - All processes see causally-related shared accesses in the same order
- **FIFO** - All processes see writes from each other in the order they were used; writes from different processes may not always be seen in that order

Explicit synchronisation

- **Weak -** Shared data is consistent only after a synchronization is done
- **Release -** Shared data is made consistent when a critical region is exited
- **Entry -** Shared data pertaining to a critical region is made consistent when a critical region is entered

Fault tolerance

Failure: whenever a resource cannot meet its promise (e.g.,CPU failure, link failure, disk failure)

- The cause of failure is known as a fault

- **Availability**: system is ready to be used immediately
- **Reliability**: system is always up
- **Safety**: failures are never catastrophic
- **Maintainability**: all failures can be repaired without users noticing (e.g., hot swapping)

Failure types

- **Crash** - A node halts, but is working correctly until it halts
- **Omission** - A node fails to respond to requests, either incoming(receive) or outgoing (send)

- **Timing** - A node's response lies outside the specified time interval
- **Response** (value, state) - A node's response is incorrect; the value is wrong, or the flow of control deviates from the correct one
- **Arbitrary** (Byzantine) - A server produces arbitrary responses at arbitrary times

Redundancy
- **Information redundancy**: error detection and recovery (mostly handled at the hardware layer)
- **Temporal redundancy**: start operation and if it does not complete start it again; make sure operation is idempotent or atomic (think transactions)
- **Physical redundancy**: add extra software and hardware and have multiple instances of data and processes

Byzantine fault tolerance - A consensus can be reached if we have 3m + 1 processes and up to m of them are faulty (2m + 1 functioning properly); the system is then Byzantine fault tolerant.

Recovery
Recovery: bringing a failing process to a correct state
- **Backward** recovery: return the system to some previous correct state and then continue
    - Take checkpoints: a consistent snapshot of the system
        - Expensive to take, need global coordination
        - When do we get rid of a checkpoint? In case of failure, how far back in time will we have to go?
    - Example: retransmission of lost packets
    - Implemented more often
- **Forward** recovery: bring the system to a correct state and then continue
    - Account for all potential errors upfront
    - For every possible error, come up with a recovery strategy
    - Apply recovery strategies and bring the system to a correct state
    - Really tricky to implement and only for specific protocols
    - Example: self-correcting codes, reconstruction of damaged packets

Virtualisation
State captures the various components of the system
- Virtual memory (physical, swap)
- Special purpose registers (program counter, conditions, interrupts)
- General purpose registers (this is the actual data that is manipulated)
- ALU floating point registers (mathematical operations)

Isomorphism
Virtualisation is the construction of an isomorphism from guest state to host state

- Guest state Si is mapped onto host state Si’ through some function V() : V(Si) = Si’
- For every transformation e() between states Si and Sj in the guest, there is a corresponding transformation e’() in the host such that e’(S’i) = S’j and V(Sj) = S’j
- Virtualisation implements V() and the translation of e() to e’()

Virtualisation monitor (or hypervisor)
- This is the actual implementation of the virtual machine
- The guest assumes complete control of the hardware
- But that is not possible — in fact, it’s a security breach
- So the monitor supervises the guest and virtualises calls to the guest’s System ISA
- Retargets them for the host
- Shares the same virtual address space with the address space it is virtualising (!)

I/O virtualisation
why:
- Uniformity and isolation
    - A disk should behave like a single local disk regardless of whether it is remote or a RAID
    - Devices isolated from one another; they operate as if they were the only device around
- Performance and multiplexing
    - Let lower-level entities optimise the I/O path; they know how to do things better than explicit read/writes
    - Parallelise the process (e.g., when replicating data)
- System evolution and reconfiguration
    - Taking the system offline to connect a new drive, or repair a damaged one is no longer an option

how?
Virtualisation through direct access

Advantages	Disadvantages
No changes to guest, same operation is what it was designed for Easy to deploy Simple monitor: only implement drivers for the virtual hardware	Cannot happen without specialised hardware Need to make the hardware interface visible to the guest: We just lost extensibility Different hardware, different drivers; Guest needs to cater for all possible drivers (not only the real ones, but the virtual ones as well!) Too much reliance on the hardware for software-related operations (e.g., scheduling, multiplexing, etc.)

Device emulation

Advantages	Disadvantages
Device isolation Stability: guest needs to operate just as before Devices can be moved freely and/or reconfigured No special hardware; all at the monitor level	The drivers need to be in the monitor or the host Potentially slow: path from guest to device is longer Possibility of duplicate effort: different drivers for the guest, different drivers for host

Paravirtualisation

Advantages	Disadvantages
Monitor now becomes simpler (and simple usually equals fast) No duplication	We still need drivers, but now drivers for the guest Bootstrapping becomes an issue: can't host a guest operating system until there are drivers available

Parallelism: Amdahl's law
Program speedup is defined by the fraction of code that can be parallelised
Fine vs. coarse granularity

Fine	**Coarse**
Low computation to communication ratio Small amounts of computation between communication stages Less opportunity for performance enhancement High communication overhead	High computation to communication ratio Large amounts of computation between communication stages More opportunity for performance enhancement Harder to balance efficiently

DBMS and DSMS

DBMS	DSMS
Model: persistent relations	**Model**: transient relations
Relation: tuple set/bag	**Relation**: tuple sequence
Data update: modifications	**Data update**: appends
Query: transient	Query: persistent
Query answer: exact	Query answer: approximate
Query evaluation: arbitrary	Query evaluation: one pass
Query plan: fixed	Query plan: adaptive

Pattern mining

Counting
Lossy
update with e^-f
at any point is E[e^f-1]

sticky sampling
we have a set S of entries of the form (e,f) [e is the element, f is the estimated frequency]
sampling an element with the rate r means we update the frequency of that element with probability 1/r
for every incoming element e:

```
    if an entry of the form (e,f) exists we increment f= f+1
    else
        we sample e with rate r
        if e is selected by sampling we add (e,1) to S
        else we ignore e
```

- ba good give-away i think: "We call our algorithm Sticky Sampling because it sweeps over the stream like a magnet, attracting all elements which already have an entry in S."

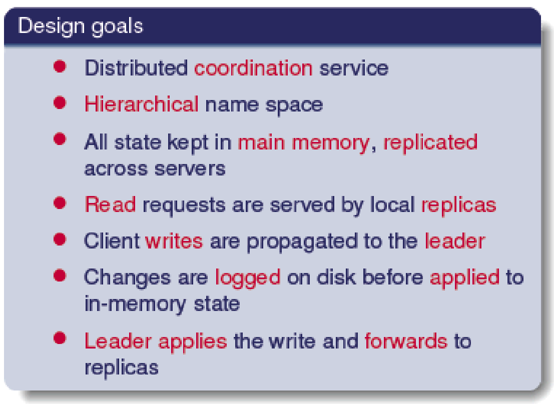
There’s a coin toss at the end of each window for each item in the data set, and it is decremented if it loses the toss.

Lossy vs Sticky:
Lossy is a lot more accurate, but sticky is more space efficient for large streams.

Low Latency processing
Storm - started at twitter
Stream
spouts
bolts

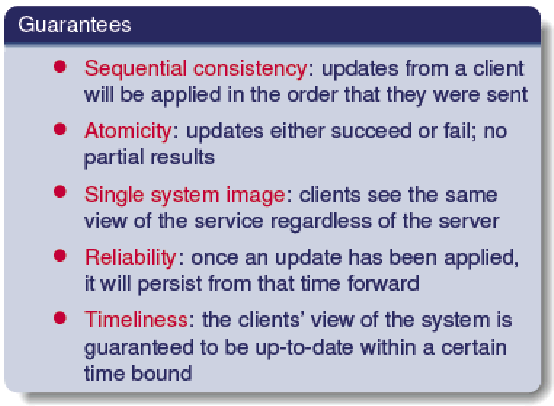
Nimbus is the master node in storm

Zookeeper - used as storage in storm



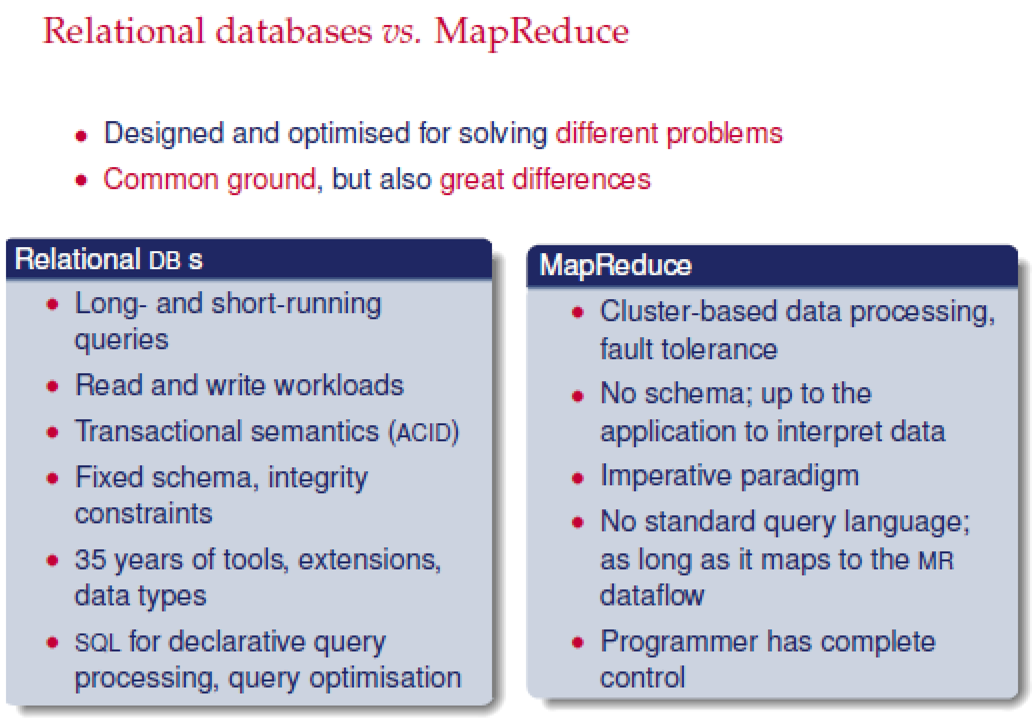
Design goals

- Distributed coordination service
- Hierarchical name space
- All state kept in main memory, replicated across servers
- Read requests are served by local replicas
- Client writes are propagated to the leader
- Changes are logged on disk before applied to in-memory state
- Leader applies the write and forwards to replicas



Guarantees

- Sequential consistency: updates from a client will be applied in the order that they were sent
- Atomicity: updates either succeed or fail; no partial results
- Single system image: clients see the same view of the service regardless of the server
- Reliability: once an update has been applied, it will persist from that time forward
- Timeliness: the clients' view of the system is guaranteed to be up-to-date within a certain time bound



Relational databases *vs.* MapReduce

- Designed and optimised for solving different problems
- Common ground, but also great differences

Relational DB s

- Long- and short-running queries
- Read and write workloads
- Transactional semantics (ACID)
- Fixed schema, integrity constraints
- 35 years of tools, extensions, data types
- SQL for declarative query processing, query optimisation

MapReduce

- Cluster-based data processing, fault tolerance
- No schema; up to the application to interpret data
- Imperative paradigm
- No standard query language; as long as it maps to the MR dataflow
- Programmer has complete control

Online transaction processing (OLTP)
Real-time, low latency, highly-concurrent
Relatively small set of fixed transactional queries

Data access pattern: random reads, updates, writes (involving relatively
small amounts of data)
Online analytical processing (OLAP)
Batch workloads, less concurrency
Complex long-running analytical queries, often ad-hoc
Data access pattern: table scans, large amounts of data involved per
query
Typically, organisations use two DB instances
OLTP frontend -> OLAP backend
Frontend optimised for transactions, backend optimised for analytics

Mapreduce 3 types of join

- in-memory join
- map-side join
- reducer-side join

DHT
Chord how to join and delete a node

Bigtable - similar to gfs
Master Tablet-Tablets-SStable

Hive data warehousing similar to hadoop built by facebook
Pig large scale data processing similar to hive tailored towards db like setting is much more efficient