Fundamentals

Key Characteristics CC: On-demand, Ubiquitous network access, Resource pooling, Elasticity, Measured Service
Concerns: Maturity, Standards, security, interoperability, control
Grid computing: solve single problem, cloud comp: solve independent tasks
Elasticity/Scalability: adapt resources, based on user's requirements
Provisioning: provide resources to user, self-service allows customer to request what they need
Virtualization: used to decouple software from hardware, partitioning(multiple apps on one machine), isolation(vm have no information about each other), encapsulation(state of vm is hw-independent)
Containers(docker) don't bundle a full OS but only libraries for an application, Hypervisor enables vm's
Multitenancy: user's share resources
Billing via metering(number of users, capacity, services)
Service Models: IaaS: computing i. as a service(Amazon ec2)
PaaS: offer platform where to run applications(facebook, google app engine)
SaaS: offer application(dropbox)
Serverless Computing: User write's code and server administration is problem of cloud provider
Cloud Deployment: Public, Private, Community, Hybrid, (virtual private cloud)
Security risks: provider has crucial part->out of control, difficult to monitor, Vm's are vulnerable

Overview Hadoop

BD Chall: Disk failures, data dependencies
Hadoop: reliable shared storage(HDFS) and analysis(MapReduce) system.
Map (k1,v1)->list(k2,v2) || Reduce (k2,list(v2)) -> list(k3,v3)
"Group by" operation that creates input for reduce is called "Shuffle and sort"
2 Types of Nodes, master(coordination), worker/slaves(run tasks/report)
Split size of input data: HDFS block size(def 128MB)
Data locality optimization: run task on a node where the data resides(only works for map tasks)
Fault tolerance: rerun task if it fails, op of map local, op of reduce HDFS
Partition: how to partition map outp
Combiner: intermediate step between map and reduce
HDFS prop: very large files, batch processing(data doesn't change), low-cost hw(expect failures), move computation rather than data
HDFS has master/slave architecture, namenode(coordination), slave(storage, read/write)
HDFS arranges data in large (128MB) blocks(less disk seeks)
Namespace is handled at namenode
Blocks are replicated among multiple slaves for fault tolerance, Namenode is responsible for ensuring replication
Rack-aware placement: two replicas in different nodes in the local rack and another one in a different rack
Read request to closest rack from reader
Communication via TCP
If Namenode fails we need manual intervention
HDFS Read: 1.client opens file through fs obj 2.fs obj calls namenode to determine location of data 3.client gets information and calls read function through input stream 4.client gets data of block, recalls read for next block 5. close in stream when finished reading
HDFS Write: Start is same, then client writes to out stream. The outstream makes packets, keeps them in a data queue and streams them to the first datanode. The datanode forwards it to the next closest datanode (recursively), once it reaches the last datanode ack, the packet. Outputstream keeps track of acked packets. After that client closes and contacts namenode to inform about datachange
YARN(Yet another recource negotiator): decouple programming model and resource management
Client requests resource manager for application master->application master requests more resources if necessary

MapReduce Programming

Anatomy of Mapreduce Job: 1.run Job 2. client ask YARN resource manager for jobID 3. client copies resources to fs 4. client tells RM to start job 5.RM allocate containter to run application(MapReduce master) in NodeManager 6.MR-Master keeps track of job progress and retrieves input splits form fs 7. Appl. master requests more container(YARNChild) for MR-tasks.
Streaming: special kind of MR for communication with I/O streams
If master fails, resource manager creates new master (same with node manager), if RM fails we fucked
Shuffle and sort happens on Map(make groups) and Reduce side(copy, sort)

MapReduce Algorithm Design

In-mapper combining -> reduce mapper outputs
Pairs/stripes -> store more information in key/values
Order inversion -> define order of keys to use this information (for relative frequencies f.i.)
Secondary sorting -> Use pairs as key to sort for value (value-to-key)

MapReduce applications

Web search problem: Crawling(gather content), Indexing(search struct), Retrieval(rank documents)
Inverted index: data struct, that provides access to list of documents that contain the key. Entry(Posting) for each key consists of a list of (doc id, payload). Can be created with MR. Use value-to-key to let mapreduce do the sorting(partitioner is used to send all original keys to same reducer
Retrieval: use distributed retrieval(distribute based on term or document) instead of MR
Shortest path with MR: parallel bfs. Needs multiple MR its.

Key-Value stores

Distributed RelatioanlDBMS via 2 phase commit: 1. Coordinator sends Vote-request, participant responds with V-commit(do comp)/V-abort 2. Coordinator sends Global-commit, if all partic. commited, else G-abort, participants handle accordingly
NoSQL: simple, flexible but not ACID. Multiple types: Key-value, column, document, Graph
Key-Value: stores pairs, key often simple, value complex. Supports Get and Put
BigTable: not fully relational, data index via row & cols, client controlls locality
BT consists of GFS(storage), Chubby(lock manager, fault-tolerance) and SSTable(file format)
SSTable: Persistent, ordered, immutable map from key to value. Consists of sequence of blocks + index for lookup
Tablet: Dynamically partitioned range of rows, consists of multiple sstables. Tablets can share sstables
Table: multiple tablets make table
BT has single master and multiple tablet servers. Coordination via master, tablet server manages set of tablets, splits them if they grow too large
Write: update log, store info in memtable first and flush it to sstable when it gets older
Read: check memtable and sstables
Compaction: minor->convert memtable to sstable, merge->merge sstables, major->left with 1 sstable
HBase: OS clone of BT, HFile SSTable, columns are grouped into families, tables partitioned into regions, uses Zookeeper for locking
HBase is distributed, col-oriented, scalable
RDBMS fixed-schema, row-oriented, strong consistency, complex queries
CAP: Consistency, Availability, Partition tolerance (only 2 out of 3 possible). CPDDB, APDNS, CASingle-nodeDB
Amazon Dynamo: highly available key-value storage, tries to be always writable, decentralized, distr. hash table, enforces eventual consistency(weaker)
Dynamo solves Partitioning(consistent hashing) and temporary failures(sloppy quorum)
Consistent hashing, map obj to nodes(storage) via hash function, replicate to following nodes, new node takes data from successor (update routing of it)
Eventual consistency via W replicas that need to acknowledge the update(not all!), eventually all N replicas update. Read contacts R servers and checks for consistency
Common (N,R,W) (3,2,2)
Sloppy quorum: read/write on the first N healthy nodes(not always the first N in the hash ring). Hinted handoff: "backup" nodes send replica to orderly node once they recovered -> eventually the first N contain the replica

Deduplication

Cloud storage pricing based on: space, requests, transfers (Tiers)
Dropbox uses deduplication for storage
Deduplication: eliminate redundant data, coarse-grained compression in chunks
Compression: transform data in another representation to safe space, fine-grained
Indicate chunks via fingerprint
Chunking: fixed/variable-sized, anchor points via Rabin-Karp algo
$p_0 = (\sum_{i=1}^{m} t_i * 10^{m-i})\%q$
$p_s = (10 * (p_{s-1} - 10^{m-1} * t_s) + t_{s+m})\%q$
Duplicates can be checked directly or out-of-order
Store Index struct: RAM(too small), disk(slow), or via bloom filter
Bloom Filter $P[FP] = (1 - (1 - \frac{1}{m})^{kn})^k$
Bloom filter is in memory and full index on disk
Sparse indexing introduces new level of granularity(segments, multiple chunks) & tries to make use of spatial locality of chunks
Extreme binning: exploits file similarity
Fragmentation happens when we get a lot of updates in the data (destroys locality)
Containers: group of chunks, basic unit of I/O
Capping: limits number of containers a segment can refer to, limit's I/O for a file
Convergent encryption(security sol): encrpyt on per-chunk basis, use hash value as crypt. key, vulnerable to offline brute-force dictionary attack
Can use server-aid for encryption
Side-channel attacks: upload file and see if deduplication occurs -> gives information about copy of file
3 problems for DropB: Hash value manipulation, stolen host id, direct up/download

Facebook photo storage

Design goals: Throughput, fault tolerant, cost effective, simple
Use content delivery networks for large-scale cache, effective because old content is rarely revisited
Haystack: offload CDN with haystack cache
Haystack directory provides mapping from logical to physical, does load balancing, checks if we go to cdn or haystack, manages rw
Pic URL: http://<CDN>/<Cache>/<Machine id>/<Logical volume, Photo>
Haystack cache gets only updated, when the request comes from the browser directly & phto is fetched from a write-enabled store machine
Haystack cache consists of store file(all the "needles"/pictures) and index file(metadata of "needles")
Store manages multpile physical volumes, handles Read(1I/O), Write(2I/O) and Delete
Write does always add 4 images to the store file, need to update index as well
Summary HS: Reduced I/O, Simplified metadata, single photo serving and storage layer
f4 Fault Tolerance: try to achieve storage-efficient fault tolerance
4 Types of Failures: Drive, Host, Rack, Datacenter(bad)
Erasure Coding: divide file into k data blocks, encode datab to additional n-k parity blocks, distribute n data/parity blocks to n nodes, any k out of n blocks can recover file data
Most common version Reed-solomon. f4: (n,k)=(14,10)
Tolerate data center failures: Double replication(data in multiple centers) or Geo-distributed XOR(add an XOR parity to two parts)
f4 is used for BLOB(big binary files), very useful for warm data

ZooKeeper

Zookeeper: highly-available/-performance coordination service. Scalable, distributed..
Guarantees: never detect old data, get informed if data changes while watching it
Zab protocol: 1.Elect leader(one of the machines that store data)2.atomic broadcast(all write request to follower will be forwarded to leader, he broadcasts it to all followers)
Data model: znode organises hierarchy like directory(intermediate nodes$\bar{=}$groups, leaf nodes$\bar{=}$members. Znode can store data
Groups are persistent, members get deleted if not accessed recently
Watches notifies client when znode content changes (only once after you need to reassign)
when a client creates a znode he takes the lock. Other clients watch the node and race the lock once it get's freed
Create ordering for lock acquisition via EPHEMERAL_SEQUENTIAL(let clients watch only the node right before them)
Create distributed barrier by waiting until a certain number of children are created(watcher on all children, then check count, if count == barrier continue)
Similarly create a producer-consumer queue. Producer creates sequential children, consumer watches all children, processes them and deletes them if there are any
Always elect the leader with the smalles seq number

MR App part 2

PageRank: measure of webpage quality.
$PR(x) = \alpha * (\frac{1}{n}) + (1 - \alpha \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$ We need to add dangling mass equally to all nodes as well.
MR needs efficient primitives for data sharing and reuse. Distributed memory needs to be fault-tolerant and efficient -> Resilient Distributed DB(read-only, partition collection of records)
Spark: driver/worker, transform(new rdd)/act(info abt rdd) on RDD, computes lazy, control of partition and persistence
Fault-recovery of rdd's based on rdd before transformation

Tail Tolerance

Tail: prob that latency is high. If tail is big overall performance gets bottled strongly
Latency is very variable due to: Resource sharing, background jobs, queueing, energy management
Reduce variability: differentiate service classes & queuing, reduce head-of-line blocking, manage background activities and synchr. disruption
Tactics for short-term adaptations: Hedged request: request data from multiple servers shortly after one another. Tied request: request data simultaneously from multiple ones
Long-term adaptations: Micro-partitions(finer-grained), selective replication(replicate hot data), latency-induced probations(reduce requests to slow server)
Special cases: Good enough(accept inclompete result), canary request(send request to a few servers and call more if successful)
Tail latency is not uncommon in storage as well, independent of I/O rate and size imbalance
How to be tail-tolerant: Reactive(extra reads, if we don't get a result in time), proactive(extra reads from the start), adaptive(detect patterns and use proactive policy)

Containerization and Serverless Computing

Containers are more lightweight than VMs (micro-service compute model). Only focus on application and its images
Docker: Daemon(coordinator), client and registry. Image contains all necessary files and has multiple read-only layers(file subset). Daemon creates writable layer out of image. Graph driver provides mapping to storage
Kubernetes: Groups containers based on purpose(POD). Master coordinates, worker run's PODs
Serverless Computing: User writes code, server provisioning and administration is offered by cloud provider. FaaS+Baas. Offers high elasticity and fine-grained resource billing
The tricky point is how to share data between jobs efficiently: common data store, Pocket
Pocket: way to store intermediate data. Uses Apache Crail, ReFlex and Kubernetes