Transactions, ACID, CAP

1. Introduction: Independent modifications to a system that stores data (Database, filesystem)
    - Several changes at once are considered single modification to the system.
2. ACID: A set of properties to implement transactions. It makes it easier to think how the system behaves.
    - Atomicity: each transaction is an independent unit which either succeed or fails. (Exp: if the money is taken from my account it gets to it destination)
    - Consistency (correctness): The system remains in a valid state. (All accounts have non-negative balance.)
    - Isolation: Even if the transactions run concurrently, the system sees them as sequential.
    - Durability: Even in the case of failure, the results of the transaction is not lost.
3. The CAP Theorem: In a system you cannot have all three consistency, Availability, and Partition Tolerance.
    - Consistency: Every read, receives the most recent write or error.
    - Availability: Every request, receives a non-error respond.
    - Partition Tolerance: The system works even if an arbitrary number of messages between the nodes in our distributed system drops.
4. The easy proof: Suppose a system is partitioned in two parts, G1 and G2: no communication happens between them
    - A write happens in G1
    - A read happens in G2
    - The result of the write is not accessible in G2, so one of these happens:
        - The system returns an error (we lose availability)
        - The system returns old data (we lose consistency)
        - The system doesn't respond (we lose partition tolerance)
    - In any distributed system you have a trade-off:
        - Either part of your system will be offline until the network partition is resolved.
        - Or you have to live with inconsistent and stale data

Consensus

1. Deterministic State Machine:
    - Each machine has a state the determines it future behavior (memory, configuration).
    - Each machine has the same starting state.
    - Receiving input changes, the state deterministically.
    - Hence two machines are in the same state if they've received the same input in the same order.
2. Read-intensive workload:
    - Input as a list(log) of commands.
    - In many architectures, most commands don't change the state:
        - Read file from distributed system.
        - Perform a database query.
        - Get a page.
    - Our distributed system can be based on servers that:
        - Work like deterministic machines.
        - Put a write command in shared log and then all the machines execute it.
        - Run read commands locally
    - If most of the workload is read then our system scale well.
3. What's Paxos about?
    - Consensus: Creating a shared log in an asynchronous distributed system solve by multi-Paxos
    - The basic Paxos algorithm finds consensus on one log entry.
    - Multi-Paxos: Agree on several things in an efficient fashion rather than running Paxos on every entry
4. How does Paxos works?
    - You want to decide with your friends to go for Pizza or Sushi.
        - you want to all agree and go to the same place.
    - Works like this:
        - Send to all "Hey, what are you doing tonight?"
        - If most of your friends answered and nobody has plans, propose pizza and send to all.
        - If most of your friends agree, then it's decided.
5. Majority wins: we want to tolerate n failures so we need 2n+1 servers.
    - A decision is taken when a majority of n+1 nodes agree on it.
        - Key reason: Two majorities must intersect.
        - Hence, at least on participant will see the conflict and avoid it.
6. Basic Paxos: Identify which proposal we're talking about.
    - Round number: we have a counter that counts the number of rounds then to identify it from others, we add our server number to it -> 42 round, server 1 -> 42.1
    - Example (we have S1, S2, and S3, value (S, P) // Pizza or sushi:
        - S1 sends a message to S2 and S3 to be prepared for a new record to start.
        - In S2 and S3 if the new N is bigger than the old one, the round number is updated and sends a promise "I'm listening" otherwise catch up to our N were at the latest N.
7. Different Scenarios of Paxos Failure:

- Acceptor in promise phase:
    - S1 sends the promise to S2 and S3 but S3 fails to send the promise. No problem will occur as long as the majority don't fail.
    - The other two continue between themselves. They still send messages that S3 might get them later.
- Proposer in prepare phase: Another proposer will take over.
    - At some point the procedure will restart and another proposer will take S1 place (S2). Servers know about round 42 so they will start a new round 43 and continue their own round (S2 proposes pizza S3 promise it and finally both accept it.
- Proposer in accept phase: Another process finishes the job.
    - S1 sends accept to S3 but fails to send accept to S2. S2 doesn't receive anything so they start a new round and chose something different. S2 becomes the proposer and sees there are servers who have already received and accepted a decision. So S2 will take the last decision that has been sent to it.

8. What could go wrong?
    - If you have one proposer:
        - One or more acceptors fail, it still works as long as majority is up.
    - If proposer fails in prepare phase:
        - Nothing happens, another proposer should eventually show up and start the algorithm.
    - If proposer fails in accept phase:
        - Either another proposer overwrites the decision
        - Or it gets notified that some other nodes have accepted the last decision, so it finishes the last job.
    - If there are two or more proposers:
        - The algorithm will never result in wrong output
        - But it will end up in infinite loop of messages
9. In the real world:
    - Leader election: Only have one proposer at a time
    - Multi-Paxos:
        - Build a full log of decisions
        - Additional complexity
        - One round trip per transaction
    - Cluster management
    - Developing, Testing and debugging
10. The cost of consensus:
    - Very difficult to implement and use correctly
    - All machines are responsible for consensus risk being bottlenecks
11. Leader election: Not just making decisions, but a full log. The roles are: Leading, Following, and Candidate
    - Heartbeats: messages with only purpose being "Hey, I'm alive" since we can only send messages in distributed, we have no way of knowing that the leader is dead or just the message didn't arrive.
    - The leader is responsible for finding and counting the votes.

- Everybody knows the number of nodes.
- Leader election scenarios:
    - 1 candidate 2 followers: a node declare itself as a candidates and requests other nodes ok, followers then by sending OK, make the candidate, the leader, finally the selected leader sends heartbeat to followers.
    - 2 candidates, 2 followers: both nodes request their closest nodes ok and those nodes, grant their ok. Again, candidates request to other followers, but they have already given their ok to other candidates so they deny their requests. Eventually with no majority the system will restart, and one of the candidates wakes up sooner and gain the majority of votes.
- Log Replication:
    -
    -

Zookeeper

1. How ZooKeeper is used (Design Goals):
    - Multiple Outstanding Requests: It designed to handle concurrent requests efficiently, this ensures scalability and responsiveness.
    - Read-intensive workload: Best suited for environment where changes are rare but access to current state is frequent.
    - General: for variety of tasks like leader election, configuration management, synchronization.
    - Reliable: It continues to work even in case of partial system failure.
    - Easy to use: Simple API for developers to interact with.
2. Building Blocks:
    - Built upon consensus algorithm: it uses ZAB its similar to Paxos, achieve consensus across nodes. It ensures strong consistency meaning nodes agree on the state of the system.
    - Wait-free architecture: Avoid locks or mechanism that block the process, ensure that the operation is in progress even in heavy load or partial failure.
        - Writers are linearizable: it ensures that all write operations appear to be executed atomically and in the exact order they were submitted.
        - Readers are serializable: Reads may lag behind write meaning they may arrive later and be outdated. However, they still follow a serial order ensuring isolation.
    - Ordering: Operations in ZK are performed under strict sequence.
    - Change Events: It allow clients subscribe to changes, allowing real-time notification when data is updated.
        - Clients can request to be notified of changes: this helps building adaptive system that reacts to changes dynamically.
        - They get notification of the change before seeing the result.
3. Data Model:
    - Hierarchical name space
    - Znodes are like both file and folder they have data and children
    - Data is read and write in its entirety

4. Create Flags:
   - Ephemeral: An ephemeral znode is a node that only exist as long as its session that created it is active. When the client (creator) disconnects or its session expires, the ephemeral znode is automatically deleted by ZK.
   - Sequence: append a monotonical-increasing counter:
     - If you create a znode with name /locks/s, the actual name will be /locks/s-1 or s-2 depending on the current sequence counter.
     - Sequence numbers are useful for task queues: ensuring tasks processed in order.
5. Configurations: A cluster requires all the nodes to follow a consistent configuration. ZK ensures that configuration changes are applied dynamically and consistently.
   - A worker starts and gets the configuration:
     - GetData ("myApp/config", watch=True)
   - Admin changes the configuration:
     - setData ("myApp/config", newConf)
   - Workers get notified of the changes and rerun getData
6. Group membership: In distributed system all nodes must register themselves and discover other nodes for load balancing and collaboration.
   - A worker starts and register itself in the group:
     - Create ("myApp/workers/" + my_name, my_info, ephemeral=True)
     - Setting ephemeral true ensure that if worker fail or get disconnected, the znode will be deleted.
   - List members:
     - getChildren ("myApp/workers", watch=True)
     - worker1
     - worker2
7. Leader election: It crucial and it has the role to manage the tasks, coordinate resources, and maintain consistency.
   - Check who's the current leader:
     - getData ("myApp/workers/leader", watch=True)
     - if successful, follow the leader
   - otherwise, propose yourself as the candidate:
     - create ("myApp/workers/leader", my_name, ephemeral=True)
     - if successful you're the leader otherwise restart
   - Since workers are watching changes for the leader, the know if the leader has changed or fails.
8. Shared locks Vs. Exclusive lock:
   - Shared locks:
     - Multiple shared locks can coexist if no exclusive lock exist with lower sequence number
     - Useful for read-only operations where multiple processes can safely access the resource.
   - Exclusive locks:
     - Only one exclusive lock is allowed and it blocks both shared and other exclusive locks with higher sequence numbers.

        - Useful for write operations where access to the resources should be exclusive.
    - Example: s-11, s-20, s-21 are shared locks and x-19, x-22 are exclusive locks.
        - Suppose s-20 is trying to acquire a shared lock:
            - It checks all nodes with lower sequence number (x-19, s11).
            - Since x-19 is exclusive, s-20 must wait until x-19 is deleted.
            - If x-19 is deleted, s-20 proceed to acquire the lock.
9. ZooKeeper Architecture:
    - All servers have a full copy of the state in their memory
    - Uses a consensus protocol that's similar to Raft
    - There's a leader that ensures consistency and propagates the updates
    - Updates are committed when majority of the servers saved the changes:
        - We use 2m+1 servers to tolerate m failures.
10. ZooKeeper Reads and Writes:
    - Reads:
        - These operations can be handled by any server in ZK cluster since all nodes have the complete copy of the data.
        - This makes read highly efficient and scalable.
    - Writes:
        - Write operation are sent to leader server. The leader server coordinates write operations by:
            i. Proposing change to follower servers.
            ii. Waiting for quorum (majority) of followers to acknowledge the update.
            iii. Committing the updates once the quorum is reached
11. Operations per second: ZooKeeper is optimized for read-heavy workloads.
    - Reads are fast because they can be served directly from any server of the cluster in-memory state.
    - Larger clusters provide greater fault tolerance but slightly reduce throughput, due to the increased communication overhead required for consensus.

Google Cloud Spanner

1. Google's problem:
    - Google engineers have several datacenters distributed across the world:
        - Sharded, replicated databases.
    - They wanted linearization (external consistency) transaction are processed in the same order they are seen in real-world.
    - Their goal was to build a whole log of all transaction similar to Paxos/Raft but because of the large scale of their clusters it will result in latency in network.
        - We can't go through Paxos/Raft since each update sent to all the nodes.
    - Solution: use Paxos locally and then resort to clock. This hybrid approach balances scalability and consistency.
    - Imagine two datacenters processing transactions. Paxos ensures local consistency within each data center. True time synchronization clocks globally, ensuring transactions are ordered correctly across.
2. Historical View:

- Core Idea: All data operations are annotated with exact timestamps; this allows the system to track:
    - When data entered the system.
- When a transaction is committed, its timestamp reflects the precise moment of commitment, minimizing the ambiguity. This enables consistent global ordering of transactions.
- To do this we need a super-precise clock. Spanner will wait to handle uncertainties in the clock.

3. True Time: ensure all servers agree on a time range, applications use these ranges to avoid conflict.
    - Bounded uncertainty:
        - TrueTime.show(): returns an interval showing the range of [earliest, latest] meaning time is between earliest and latest.
    - Time Masters: each datacenter has a set of Time Masters machines:
        - Regular ones, GPS clocks
        - Armageddon master, atomic clocks
4. Talking to a time master: the system synchronizes with the time master using method similar to Network Time Protocol (NTP).
    - Variables:
        - T0 the moment client request to server.
        - T1 server receives the request from client.
        - T2 server sends the respond to the client.
        - T3 client receives the respond from server.
    - Time offset: $\theta = \frac{(t_1 - t_0) + (t_3 - t_2)}{2}$
    - Round-trip delay = $\delta = (t_3 - t_0) - (t_2 - t_1)$
    - Measurements are described before is repeated, and statistics are extracted.
    - The output is an interval.
    - Some Time Masters may provide incorrect data ("Liars"). An algorithm is needed.
5. Intersection Algorithm: This technique is critical in distributed systems to resolve timing discrepancies and ensure consistency. It helps identify most reliable time intervals based on the consensus across multiple sources.
    - Create a list containing (a, +1) and (b, -1) pairs for each [a, b] interval and sort them by first element. We call them (v_i, d_i).
    - Compute the cumulative sum s_i for all , d_j values where $i \leq j$.
        - It's the number of intervals overlapping in $[v_i, v_{i+1}]$.
    - Find the maximum value the result will be $[v_M, v_{M+1}]$.
6. Catering for Local Clock Error:
    - After the clock is synchronized, the uncertainty grows according to the worst-case assumptions on the computers' clock drift.
    - After synchronization, the clock's uncertainty begins at a small reference (+- 6 microseconds) over time, this uncertainty increases due to clock drift.
    - The drift rate is given 200 microseconds per seconds. Clock drift occurs because no clock is perfectly accurate. Small deviations in frequency cause time discrepancies.

7. Version Management: Versioning proper ordering and visibility updates in distributed systems.

Time	My friends	X’s friends	My Post
4	[X]	[me]	
8	[]	[]	
15			[“Government bad”]

- In this example:
    - At time 4 X is friend.
    - At time 8 X is removed from friends list
    - At time 15 a post is made
- Vanishing and timestamps:
    - Each update (write) is assigned a timestamp.
    - Reads are associated with specific timestamps, ensuring they only see updates committed before the timestamp.
- Even if a write transaction happens in completely different cluster, since they are globally ordered, consistency is ensured.

8. Data Model:
- A key-value store:
    - Lookup the value for a given string, as if it was a huge hashtable.
    - Holds data like: (key:string, timestamp: int64) -> string
- Nodes responsible of a key in multiple continents
    - Use Paxos to get consensus
- Allowing asking the value at a given moment in time
    - SQL-like semantic added forwards

9. Assigning Timestamps to Writes:
- Transaction begins
- Lock acquired: when transaction begins it acquires lock to prevent conflicting updates.
    - The timestamp is assigned during the lock phase
    - Lock ensures no other transaction modifies the data until the current transaction is completed.
- Lock released: Once the lock is released, the transaction timestamp represents the commit time.
- Transaction completed.

10. Transaction conflict: Two transactions are conflicting if they operate on the same data concurrently.
- T1 starts and lock the data
- T2 starts before T1 finishes it but cannot lock the same data until the T1 release it.
- T2 sees the state of the data as it was before T1.

11. Non-conflicting Transactions: if two transactions operate on different data, they can execute in parallel.
    - T3 and T4 modify different keys so they don't interfere with each other.
    - Timestamps are assigned based on when the transactions begin.
12. True time to assign Timestamps
    - Phases:
        - Lock acquired
        - ts = TT.show().latest
        - Work done
        - TT.now().earliest > ts
        - Lock released
    - If the system is afraid of finishing before the ts, it just waits until there's no doubts.
    - If uncertainty on time is too large, the system gets slowed down.
    - Clock uncertainty should be smaller than transaction length.

Erasure Coding

1. Introduction: I have data to save (100 GB), I can store it on N different servers, I want to be able to recover my data even if M servers fail and I lose my data. How to minimize the cost?
2. Trivial Solution: replication:
    - To safeguard my self against M server failures I will copy my data on M+1 servers.
    - Redundancy: The ratio between the amount of data I store and the original size of data.
        - For M=2, I need 300 GB, Redundancy=3
3. For N=3, M=1, Parity:
    - I split my data into 2 blocks B0, and B1.
    - I create a parity redundant block BR
        - BR = B0 XOR B1
    - If I lose one of the Bi, I can recover it using parity bock:
        - Bi = Br XOR Bi-1
        - This is because: X XOR (X XOR Y) = (X XOR X) XOR Y = Y
    - Redundancy: 1.5:
        - For 100 GB I need 150 GB space
        - Only 100 GB needed to recover original data
4. For any N, M=1:
    - I split my data into K = N – 1 blocks of the same size
        - N = 6, 100 GB: 5 blocks of size 20 GB.
    - I create a parity redundant block BR:
        - BR = B1 XOR B2 XOR … XOR BN
    - If I lose one of the Bi blocks, I can recover it:
        - Bi = B1 XOR B2 XOR Bk-1 XOR BR
    - Redundancy: N/K = N/(N-1):
        - Say N=6, 100 GB: redundancy = 1.2, I need 120 GB
5. Erasure Coding Magic: any N & M:
    - I encode my data in N blocks, each of size 1/K of the original size, where K = N – M:
        - If N=6, M=2: each block is 25 GB

- Redundancy: N/K

6. Any N, M=N-2: Linear oversampling: A single straight line connects any two distinct points. Hence we can compute a and b with any two (Xi,Yi) pairs.
   - Message: a, b
     - a=3, b=2
   - f(x) = ax+b
   - Encoded message: Y0 = f(X0), Y1 = f(X1), …, Yn-1 = f(Xn-1)
     - With Xi = i: (2, 5, 8, …)
   - With any two distinct (Xi,Yi) pairs, a system of two linear equations and 2 variables:
     - With Y2 = 8 and Y5 = 17
     - aX2 + b = 8, aX5 + b = 17
     - from here, it's trivial to compute the message
7. Any N & M: Polynomial Oversampling: The message is represented as the coefficient C0, C1, …, Ck-1 of a polynomial f(X) of degree K-1
   - Any K = N – M distinct points identify a single polynomial degree K-1. Hence, we can find the message with any K (X, Y) pairs.
     - $f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$ for k = 3
   - Problem: Numbers grow in size! If we encode them as bits, number in encoded message will be bigger than those in the original one.
     - Solution: using finite fields
8. Fields: A set in which addition, multiplication, subtraction, and division are defined and behaved as in real and rational numbers.
   - Finite Fields are useful because:
     - They allow operations in encoding/decoding to be well-defined and manageable.
     - In erasure coding, field ensures that arithmetic operations work efficiently.
9. Skipped Theory and focus on practical Usage:
   - N is the number of machines that will store your data
   - M is the number of failures you want to tolerate
   - Choose p not smaller than N
   - Divide your data in K=N-M blocks of same size
   - Encode it as a series of values smaller than p
   - All elements of first block will be C0 coefficients, second block C2 and so on.
   - Encode and put all the Y0 coefficients in the first encoded block, the Y1 coefficients in the second block and so on.

Queueing and Scheduling

1. Queueing Theory: we have a single server and a queue of jobs it has to serve (webpages, computations, …)
   - λ: The average rate at which jobs arrive in queue
   - μ: The average rate at which the server process them
2. M/M/1 (Kendal notation): describing specific type of queue
   - M: Markovian (memoryless) nature of the arrival and service process
   - 1: there's only 1 server
   - Memoryless properties:

    - Arrival: the chance of a new job arriving doesn't depend on the previous jobs. Every moment is independent.
    - Service time: The time it takes to service a job is independent of how long it has already been in service. This is similar to how dice roll is independent of previous rolls.

3. Why simple model? They are simple so less computation, give insight on real world problems, and can be used as benchmarks
    - Real models are not like this but some of the insights does apply to real-world use-cases.
        - The memoryless property makes it much easier to analyze and deviate formulas.
        - The insights from analyzing simple models can be applied to complex ones.
        - The simple model serves as a benchmark to compare complex models and verifying mode detailed simulations.
4. M/M/1 Equilibrium: The system achieves equilibrium only if $\lambda < \mu$. This ensures the server can process jobs faster than they arrive.
    - Balance of probabilities: The rate of jobs moving into state i+1 must be equal to the rate of jobs moving out. $\lambda p_i = \mu p_{i+1} \rightarrow p_{i+1} = \frac{\lambda}{\mu} p_i$
    - Let's modify and say $\mu = 1$ so $p_{i+1} = \lambda p_i$.
    - Using revised relationship, the probabilities are:
        - $p_1 = \lambda p_0, p_2 = \lambda^2 p_0, p_3 = \lambda^3 p_0, \dots, p_n = \lambda^n p_0$
    - Since this is a probability of distribution, its sum must be one. We can then solve everything: $\sum_{i=0}^{\infty} \lambda^i p_0 = 1; \frac{1}{1-\lambda} p_0 = 1; p_0 = 1 - \lambda; p_i = (1-\lambda)\lambda^i$
        - The average queue length is $L = \sum_{i=0}^{\infty} i p_i = (1-\lambda) \sum_{i=0}^{\infty} \left(i\lambda^i\right) = (1-\lambda)\frac{\lambda}{(1-\lambda)^2} = \frac{\lambda}{1-\lambda}$
5. Little's Law:
    - L: average number of jobs in system
    - λ: arrival rate
    - W: average time spent in the system
    - $if\ \lambda$ is the rate that jobs arrive to the system and w is the time each job spend in the system then λw represents the total time jobs spend in the system per unit time.
    - The average time spent in the system for an M/M/1 FIFO queue is:
        - $W = \frac{1}{1-\lambda}$
6. Multiple Servers: We now introduce a more realistic scenario; having multiple servers instead of one. This extension allows to handle higher loads by distributing jobs across several servers.
    - Multiple servers: Assume n servers are available to process jobs. A load balancer is used to assign jobs to servers efficiently.
    - Load balancing randomly: each server will handle a fraction of total load, roughly λn the performance will resemble that of n separate M/M/1 queues, each with reduced load.
    - Optimized load Balancing: instead of random assignment, jobs can be assigned to the least loaded server. This minimizes waiting time and prevents bottlenecks.
7. Supermarket Queueing: Instead of querying all n servers to determine their loads and queue length, just ask a small number of d of the them and assign to the shortest queue.
    - This approach minimizes the likelihood of server being overloaded.

- By querying only few servers, the system maintains near-optimal load balancing while minimizing the overhead of checking every server.
- This strategy is especially beneficial in large scale clusters where queuing all servers is impractical.

8. Round Robin Scheduling: You run a job for a given timeslot and then pass the turn to another one:
   - If n jobs are in the queue, each will proceed at $\frac{1}{n}$ speed.
   - Often adapted with priorities reflecting job importance.
9. Shortest Remaining Process Time (SRPT): It always prioritize the job with smallest remaining process time.

Aspect	Preemptive policies	Size-based scheduling
Core idea	Pause/resume jobs to improve fairness and efficiency	Use job size to prioritize
Implementation	Round Robing with Priority	SPRT
Strength	Simple fair for all jobs	Optimal for minimizing W
Challenges	May not handle job sizes variation well	Potential for starvation for large jobs

10. Evaluating Distributed Systems:
    - Mathematical Models:
      - Good: Hard truth for modeled world
      - Bad: Need simplifications: models may not fully represent the real world
    - Experiment and measurement on real systems.
      - Good: directly evaluate the performance of real system
      - Bad: Overly focused on implementation, expensive, limited
    - Simulations:
      - Good: scalable and inexpensive, useful for modeling large and complex systems
      - Bad: Trade-off between precision and scalability, simulation may not compute all the real-world complexity

Tor

1. Tor Purpose: to secure intelligence communication
   - Privacy debate:
     - Main argument against complete privacy is Crime and Terrorism
     - Some arguments for it:
       - Protecting whistleblowers, activists, journalism
       - Avoid tracking by Governments
       - Crim already have access to anonymity
2. How it works: Each router peals a layer (removes encryption) and forwards the message to the next router.
   - The message is sent from final router without encryption to the destination.
   - No router knows both the source and destination.
   - Tor Continuously randomizes the routing paths making it challenging for attackers to trace communication.

- Example:
    - Alice’s Tor client obtains a list of Tor nodes from a directory server.
    - Alice’s Tor client picks up a random path to destination server.
    - If at a later time, the user visits another site, Alice’s Tor client select a second random path

3. Roles of nodes in Tor:
    - Guard node: The entry point to the Tor network
    - Relay node: forwards the traffic between nodes, hiding the original message
    - Exit node: The final node that forwards the decrypted source data into the destination.
    - If there’s some crime in the message, all nodes are just relaying it without knowing about the criminal content of it only the exist node sees the unencrypted message and most likely to get tracked by the law and get charged with a crime.
4. Security Assumption: to ensure anonymity:
    - Independent nodes: All routers you choose shouldn’t be owned by the same attacker.
    - Guard and exit nodes: Attackers can’t see your traffic from both guard and exit nodes. Otherwise, they can correlate traffic patterns and compromise anonymity.
5. Tor is a SOCK proxy: SOCK is a protocol that allows encapsulating TCP protocol.
    - SOCK support UDP since version 5, but Tor doesn’t support UDP connection.
    - You should only use apps that will use proxy servers.
    - BitTorrent uses UDP protocol, therefore using it over Tor isn’t a good idea for anonymity.
6. Tor Browser: Using Tor on your browser isn’t a great idea. It eliminates common tracking vulnerabilities like cookies and fingerprint.
    - Cookies: Websites tack users via cookies to corelate activities across platform like ads
    - Finger printing: websites gather specific data about user’s hardware and software configuration.
    - The Tor browser is hardened Firefox designed to look identical for all users and never exit the proxy.
7. Bridges: Tor nodes are publicly known and some countries block access to Tor’s nodes.
    - Bridges exist allow people to use Tor anyway.
    - Bridges enhance accessibility in countries or regions with restrictions on internet.
    - People can ask for access to a Tor bridge at a time:
        - You ask for a Tor bridge. And success to a single Tor bridge is sent to you. You solve a CAPCHA. This is harder to control since getting a list of bridges is harder work.
8. Onion Services Steps:
    - Bob selects introduction points and creates a circuit of them.
    - Bob advertises his service, XYZ.onion at the database.
        - The database is decentralized, (Distributed Hash Table) DHT.
    - Alice hears that XYZ.onion exists, she requests for more information from the database. She also sets up a rendezvous point.
    - Alice sends a request to Bob via the introduction point sharing the rendezvous point and a one-time secret.
    - Bob connects to Alice’s rendezvous point and provide her one-time secret.
    - Both use Tor circuit for communication.

9. How Big is Tor?
   - 2M relay users:
     - 19% USA, 10% Germany, 5% South Korea
   - 130K bridge users:
     - 46% Russia, 14% Iran, 9% USA
   - 8K relays
   - 2K bridges
   - 300 GB/s aggregated bandwidth
10. Tor History and Research: Tor's effectiveness depends on proper use and continues improvements.
    - The security of Tor is not perfect.
    - However, the history tells use it's good enough in most cases.
11. Website Fingerprint: A technique to identify which website the user is looking at by looking at the size and timing of encrypted packages.
    - Tor minimizes fingerprinting with:
      - Fixed message size.
      - Latency for obfuscate patterns
    - Tor uses messages of fixed size 512 bytes (called cells)

Search in P2P Systems (30)

1. Introduction to P2P: Distributed applications where nodes (peers) play equal roles.
   - Self-organized and decentralized.
   - Fault-tolerant and censorship resistant.
   - Efficient use of resources.
   - File sharing (Napster, BitTorrent).
   - Cryptocurrency (Bitcoin, Ethereum).
2. Napster: Centralized P2P
   - Central index server maintaining files meta data.
   - Peers connected via server for file sharing.
   - Limitations:
     - Vulnerable to shutdowns.
     - Legal issues due to the copyright.
3. Gnutella: Decentralized P2P.
   - Overlay network: peers connect to fix number of neighbors.
   - Flooding search: Queries forward to all neighbors, resulting in high redundancy and bandwidth use.
   - Improvements with Ultrapeers:
     - Separate high-capacity "Ultrapeers" from low-capacity leaf nodes.
     - Efficient query routing with reduced traffic.
     - Introduced query routing tables (QRT) for more targeted searches.
4. Bloom filters: Compact probabilistic data structures for set member ship tests.
   - Add (x): Hash an element and set the corresponding index in the bit array.
   - Test (x): Check all the corresponding bits for test.
   - False-positives are possible, False-negatives are not.

- Extremely memory efficient.
- Applications:
    - Database caching (avoid unnecessary disk reads)
    - Web caching (storing most frequent requests)

5. Distributed Hash Tables (DHT): A decentralized hash table for efficient key-value lookups.
    - Structured P2P overlay: Nodes connect strategically for efficient routing.
    - Consistent Hashing: Ensures minimal disruption when nodes join/leave.
    - Redundancy: Data is replicated across multiple nodes.

6. Chord protocol: Nodes form a ring with identifiers in a predefined range ($[0, 2^m]$) and each node maintain link with predecessor and successor.
    - Finger tables: shortcuts to nodes at exponential increasing distances ($O(\log(n))$), enabling faster lookups.
    - Greedy routing: forward queries to the closest node to the target key.
    - Nodes maintain a list of successors for redundancy.
    - Periodic stabilization ensures links are updated as nodes join/leave.
7. Kademlia:
    - A popular DHT used in systems like BitTorrent.
    - Nodes Use XOR-based distance metrics for routing.
    - Logarithmic lookups complexity.
    - Symmetric links for better routing efficiency.

Questions

1. What are the key characteristics of a P2P system?

- Decentralized Architecture.
- Equal roles of each node.
- Fault tolerance and resilience to failure.
- Efficient usage of resources.

BitTorrent (10)

1. BitTorrent definition:
    - It’s a decentralized method of distributing files, often large ones, across the internet.
    - BitTorrent doesn’t have the functionality to search the files itself. Instead, users rely on external indexing services and websites to find .torrent files.
    - .torrent is a metadata file that includes the necessary information to locate and download files.
    - .torrent contains some of these features (Name, size, hashing information, Tracker URL, DHT)
2. BitTorrent Swarm: the network that contains many nodes and it separates into several parts:
    - Tracker: A server that connects peers by keeping a list of all active nodes. DHT made tracker less critical.
    - Seed: A peer with complete copy of the file, it only uploads the data to others.
    - Downloaders: Peers who have not finished downloading the file. As they download, they can upload pieces to others, contributing the swarm.

- Seeding behavior: After finishing a download, peers can remain in the network as uploaders.

3. Pieces:
   - Large files divided into smaller manageable chunks (256KB by default).
   - Each piece has a unique hash included in .torrent file. This Ensures that a downloaded piece matches the original data, avoiding corruption or tamping.
   - Piece Exchange: nodes query others to find out pieces they have. Allows efficiency.
4. Piece selection:
   - When downloading the files, BitTorrent prioritizes certain pieces to optimize distribution and availability.
   - Rarest First: This algorithm focuses on downloading the rarest pieces first. These pieces are harder to find and might disappear if peers holding them leave the swarm.
   - Random first Piece: BitTorrent may choose a random piece to download first. This allows the peer to quicky upload pieces to others, contributing to the swarm even while downloading.
5. Tit for tat:
   - Behavior strategy: I behave with you as you did with me.
   - Most upload slots are chocked, meaning the uploader doesn't send data through those connections.
   - By default, 4 connections are unchoked meaning the uploader actively sending data to them.
   - Optimistic Unchoke: For every 30 seconds, BitTorrent Unchokes one random node. This gives chance for the new node to prove their worth.

Questions

1. What is BitTorrent, and how does it differ from traditional file-sharing methods like HTTP and FTP?
   BitTorrent is a P2P file sharing system where file is divided into pieces and shared among peers. BitTorrent allows downloading from multiple peers simultaneously.

Eventual Consistency

1. Definition: Data updates propagates to all replicas eventually. Reads might temporarily see stale data.
   - Partition detection: Identifying when network partition occurs.
   - Partition mode: define system behavior during partition.
   - Recovery: Strategies to merge divergent states after partitions.
2. Amazon Dynamo: Its purpose is to provide high availability for Amazone shopping cart's service.
   - Eventually consistent with user's perceived consistency.
   - Highly capable: Handles millions of requests/day with 99.9995% availability.
   - Architecture:
     - Key-value store: Stores data as key-value pairs.
     - Consistent Hashing: Eventually distributes data and reduces rebalancing cost when nodes join/leave.
     - Replication: Data is stored on multiple nodes to ensure durability.

3. Quorum-based systems:
   - Sloppy Quorum: parameters:
     - N: Total replicas.
     - R: Minimum replicas for read.
     - W: Minimum replicas for write.
   - Guaranties:
     - R + W > N: High consistency
     - Lower R/W values improves latency but reduce consistency.
   - Example:
   - N=3, R=2, W=2: Default(balance)
   - N=x, R=1, W=x: Fast reads
   - N=x, R=x, W=1: Fast writes.
4. Conflict Resolution:
   - Vector clocks:
     - Tracks changes across nodes.
     - Resolves conflict by comparing counters:
       - If one version dominates, it supersedes other.
       - Otherwise, both versions are returned for client-side resolution.
   - Merkle Tree: Is a method of solving inconsistency when you don't have ACID properties. Each machine that has one of the N copies may have different version of them. So since every machine has the hash of everything, they start getting to the top node of the tree. If they have same node, they have averaged, otherwise they start compering children to find the root of divergent
     - Tree-based structure for efficient comparison of replica data.
     - Each node's hash summarizes the data beneath it.
     - Quickly identifies and reconciles inconsistencies.
5. Failure Handling:
   - Transition failure:
     - Reads/writes spill over to the next node during failures.
     - Updates are synced when failed nodes rejoin.
   - Anti-Entropy: Synchronizes using Merkle trees to compare and repair data.
   - Client-side reconciliation:
     - Clients handle unresolved inconsistencies.
     - Example: Amazon retains all shopping cart items in conflict scenarios.
6. Optimization:
   - Buffered Writes: Baches writes for efficiency, trading off immediate consistency.
     - Waite for few writes to be committed before writing on the disk
   - Throttling: Slows down background tasks during peak loads to improve performance.
   - Load Balancing: Routes operations to the fastest-responding nodes.
7. Cassandra:
   - Inspired by Dynamo, developed by Facebook and now under Apache.
   - Similar architecture (Consistent hashing, replication).
   - Zookeeper:
     - Leader election.

        - Rack-aware and datacenter-aware placement.
    - Conflict resolution: Uses timestamps instead of vector clocks (latest writes wins)
    - Data Migration: Lightly-loaded nodes are migrated dynamically on the ring.
8. Take aways:
    - CAP Theorem: Understands the trade-off between consistency, availability, and partition tolerance.
    - Eventual consistency: ensures high availability with eventual convergence.
    - Dynamo vs. Cassandra: Both focus on availability and scalability but differ in conflict-resolution strategies.

Sample questions

1. What is the CAP theorem, and what does it state?
   The CAP theorem states that in a distributed system, it is impossible to simultaneously guarantee Consistency, Availability, and Partition Tolerance. During a network partition, a system must choose between consistency and availability.

2. How does the ACID model differ from the BASE model?
   ACID ensures atomicity, consistency, isolation, and durability, prioritizing strict consistency and reliability. BASE (Basically Available, Soft State, Eventual Consistency) relaxes these guarantees, favoring availability and scalability with eventual consistency.

3. What does eventual consistency mean in distributed systems?
   It means that all updates eventually propagate to all replicas, and the system will converge to a consistent state over time, assuming no new updates.

4. What are the main challenges of partition recovery?
   Partition recovery involves detection and repairing inconsistencies, determining the rules of merging states, and ensuring consistent state after the partition ends.

5. How do ATMs handle partition mode?
   ATMs operate in partition mode by allowing transactions like withdrawals. They impose limits (200$ per person) to reduce risk and reconcile inconsistencies during recovery.

6. What was Dynamo designed for, and why is availability critical for it?
   Dynamo was designed for Amazon’s shopping cart service, where availability directly affects revenue. It ensures users can add items to their cart even during system failures or partitions.

7. What is the significance of consistent hashing in Dynamo?
   Consistent hashing evenly distributes data across nodes, minimizing data rebalancing when nodes join or leave, and ensures efficient partitioning.

8. What is sloppy quorum, and how does it work?
   Sloppy quorum is a configurable consistency mechanism. It ensures operations succeed even when some replicas are unavailable.

The MapReduce Paradigm

1. Big Data and its challenges: big data referrers to datasets that are too big, complex, or fast changing for traditional data processing tools often characterized by 3 Vs:

- Volume: The size of data
- Velocity: The speed that data is generated and processed
- Varity: The different types and formats of data.

In many cases, the size of the data and its inherent value often outweigh the complexity of the algorithm used to process it. This necessitates efficient frameworks like MapReduce to handle data.

2. The MapReduce programming model: it's inspired by functional programming, is a distributed computing framework designed for large-scale data processing on clusters of commodity hardware. The core principle of MapReduce:
   - Scale out, not up: It is more cost-effective to use many commodity servers that a few high-end machines
   - Shared nothing Architecture: Each process node operates independently, minimizing synchronization overhead.
   - Move processing to the data: Data is processed where it resides minimized network transfer

   Failures are common in large networks and MapReduce is built with fault tolerance in mind, ensuring that transient issues do not compromise the overall computation.

3. MapReduce workflow:
   - Map phase:
     - The input dataset is divided into chunks.
     - Each chunk is procced by a mapper, which transforms the data into key-value pairs.
   - Shuffle and sort phase:
     - Intermediate key-value pairs are grouped by key and sorted.
     - This ensures that all values associated with a particular key are processed together in the next phase.
   - Reduce phase:
     - Reduces the aggregated values associated with each key to produce the final output.

   This model is inherently parallel, making it scalable and efficient for distributed systems.

4. Functional programming roots:
   - Map: Applies a transformation function to each element of a dataset, producing a new sequence. (exp. Incrementing each value of list)
   - Reduce: Aggregate elements of sequence using a binary operation, starting with an initial value. (exp. Summing numbers in a list)

5. Combiners in MapReduce: Combiners are optimization steps in the MapReduce framework. They act as a mini-reducer, performing partial aggregation of the data locally on the mapper nodes before the shuffle phase. This reduces the amount of intermediate data transferred over the network, improving the performance.
    - Mapper emits: (“White”, 1), (“White”, 1), (“snow”, 1).
    - Combiner aggregates: (“White”, 2), (“snow”, 1).
    - Reducer processes the combined output, producing the final counts.

    While combiners optimize performance, they are not always guaranteed to run and must not affect the correctness of final result.

6. Scalability and Applications: MapReduce is designed for
    - Batch processing: handling jobs that requires full dataset scan, such as processing social graphs, training ML models.
    - Embarrassingly Parallel Problems: Tasks that can be divided into independent sub-problems, such as data format conversion or filtering.
    - Examples of applications: Analyzing data logs, NLP applications in recommender systems.

7. Advanced patters in MapReduce: Some coming patters in MapReduce are:
    - Co-Occurrence Matrices: Counting how often two items (e.g., words) appear together in a context.
    - Pairs and Stripes Approaches:
        - Pairs: Emits complex keys like (word1, word2) with associated counts.
        - Stripes: Emits a single key (e.g., word1) and a dictionary of co-occurring words.

MapReduce is a powerful framework for processing massive datasets in distributed systems. By abstracting the complexity of parallel and fault-tolerance computing, it allows developers to focus on defining map and reduce operations, making it a cornerstone of data processing systems.

Sample Questions

1. **Explain the 3 Vs of Big Data and their significance.**
    - Volum: the size of dataset.
    - Velocity: the speed of data which being generated and needed to be processed.
    - Varity: the format and type of processing data.

2. **What is the MapReduce programming model, and what are its main phases?**
    - MapReduce is a distributed computing framework inspired by functional programming; the main phases are:
        - Map Phase: transforming input data into key-value pairs.
        - Shuffle and sort: Group and organized key-value pairs based on their keys.
        - Reduce phase: Aggregates and process sorted pairs and produces final-output.

3. **Differentiate between ‘scaling out’ and ‘scaling up.’ Why is scaling out preferred in MapReduce?**
    - Instead of using small number of high-end processing units, we should acquire as much as computing commodity as possible.

- Scale up: adding resources to a single machine (RAM, CPU, …)
- Scale out: using multiple commodity servers in parallel.

4. **Why is the "Shared Nothing Architecture" important in distributed systems like MapReduce?**
   - This architecture ensures nodes operate independently, reducing the network load by less synchronization overhead, less latency, and improving fault tolerance.

5. **Describe the role of combiners in the MapReduce framework. Provide an example.**
   - They are part of the optimization step in MapReduce framework, act as a mini-reducer, aggregating part of the data on mapper node.
   - Example: wordcount, the combiner can sum counts locally.

6. **What challenges does MapReduce address in processing Big Data?**
   - Fault tolerance in large clusters.
   - Efficient use of distributed resources.
   - Handling bottlenecks like disk I/O and network transfer.
   - Scalability with increasing data size.

7. **Explain the difference between the Pairs and Stripes approaches in co-occurrence matrix computation.**
   - Pairs approach: (word1, word2) as a key with a count value, e.g., (("apple", "banana"), 1)
   - Stripes Approach: Emits word1 as a key with a dictionary of co-occurring words, e.g., ("apple", {"banana": 1, "orange": 2})

8. **How does the shuffle and sort phase contribute to the efficiency of MapReduce?**
   - It groups and organizes the key-value pairs by key, ensuring all values of a key are processed together, reducing redundant data and enabling efficient reduction.

9. **What does "moving processing to the data" mean, and why is it crucial in MapReduce?**
   - It means running computations on the nodes where data resides, minimizing data transfer over the network and reducing data processing time.

10. **Why are failures considered "the norm" in distributed systems like MapReduce, and how does the framework handle them?**
    - Failures are common due to the hardware and network issues. MapReduce handles them using fault tolerance mechanisms like data replication and executing of failed tasks.

11. **Compare sequential and random access in the context of Big Data processing. Why is sequential access preferred?**
    - Sequential Access: Reads data in a continuous stream, making it faster.
    - Random Access: Accesses scattered data, leading to slower disk seeks.

12. What scalability goals does MapReduce aim to achieve?
    - Doubling the data should ideally doubling the process
    - Doubling the cluster size should halve the process time.

13. **What is the role of the distributed filesystem in MapReduce?**
    - It provides reliable storage, supports large files, ensures data locality, and replicates data to ensure fault tolerance in distributed environments.

Apache Hadoop

1. Introduction to Apache Hadoop: It's a widely adopted framework for distributed storage and processing large datasets. It's particularly useful for organizations lacking proprietary solutions like Google's internal tools. The primary components of Hadoop are:
   - HDFS (Hadoop Distribution File System): a file system designed for distributed storage.
   - MapReduce: A programming model for distributed systems.
2. HDFS: is the backbone of Hadoop, designed for massive datasets across multiple machines and enables computation near the data. Inspired by google file system (GFS), HDFS focuses on throughput rather than latency, optimizing for read-heavy workloads and operating on commodity hardware to reduce the cost.

   Key features:
     - Large datasets: data is split into blocks typically 128 MB each.
     - Fault tolerance: Data blocks are replicated across multiple machines (3 DataNodes by default).
     - Read-intensive design: sequential reads are prioritized over low-latency access.
3. HDFS blocks: Data in HDFS is divided into several chunks(blocks), which:
   - Improve seek time: large block sizes reduce the impact of seek times compared to read times.
   - Simplify Metadata Management: Large blocks means fewer meta data entries, making it efficient to store metadata in the NameNode's memory.

   Blocks are replicated to ensure fault tolerance and allow processing to begin immediately without waiting for reconstruction, as erasure coding would require.

4. HDFS Architecture: it consists of three primary node types:
   - NameNode:
     - Stores meta data in RAM, including the directory tree and block index.
     - Handles updates synchronously, ensuring data consistency.
     - Uses a journal to log changes, often stored on a network for reliability.
   - DataNode: Store the actual data and periodically reports block status to the NameNode.
   - Secondary NameNode:
     - Keeps a copy of NameNode's edit log.
     - Can act as a read-only backup if the NameNode fails.
5. File Read and Write process:
   - File Read:
     - The client queries the NameNode for block locations.
     - Data is retrieved from the closest DataNode to minimize the latency
   - File Write:
     - The client requests a set of DataNodes for replication (default: three nodes).
     - Data is written in pipeline, with each node replicating to the next.
     - Replicas are placed across racks for reliability and bandwidth efficiency.

6. Scheduling in Hadoop: A Hadoop job is divided into independent tasks to maximize parallelism. These tasks are managed using various scheduling strategies:
   - FIFO Scheduler:
     - Prioritizes the tasks based on arrival time.
     - May cause smaller jobs to starve when large jobs dominate the queue.
   - Fair Scheduler:
     - Ensures each jobs gets a fair share of the resources.
     - Dynamical balances the workload across active jobs.
   - Capacity Scheduler:
     - Divide resources into virtual clusters with dedicated queues.
     - Allows elasticity by reallocating unused resources.
7. Map Reduced Tasks:
   - Map Tasks:
     - Process one input split (default: one HDFS block) per task.
     - Schedular prioritizes running tasks on nodes where the data resides.
   - Reduce Task:
     - Aggregate data from map task.
     - Allow custom partitioning to handle skewed datasets effectively.
8. Shuffle and sort phase: the shuffle phase bridges map and reduce tasks by redistributing data across nodes.
   - Map side:
     - Outputs are buffered in memory and written in disk in the sorted order when full.
     - Combiners can be used to reduced data before transmission.
   - Reduce side:
     - Reducers fetch data from mappers and merge it using distributed mergesort.
     - Outputs are saved back in the HDFS replication.

9. Handling Failures: Fault tolerance is a core feature in Hadoop:
   - Task Failures:
     - Tasks are retired up to four times by default.
     - Hanging tasks are killed and retired.
   - Node Failure: The NameNode detects missing heartbeats and removes failed nodes from the cluster.
   - Scheduler Failure: Zookeeper can be employed to maintain a backup of the scheduler.

Hadoop design efficiently manages distributed storage and computation through HDFS and MapReduce. Its robust architecture, fault tolerance, and versatile scheduling strategies making it a powerful tool for big data applications. By breaking the task into manageable units and ensuring computations occur near the data, Hadoop minimizes overhead and maximizes performance, making it a critical framework in the field of distributed computing.

Sample Questions

1. **What are the primary components of Hadoop, and what's their role?**
   - HDFS: Distributed storage system, designed for handling large datasets.
   - MapReduce: Programming model for parallel processing of data.

2. **Why does HDFS prioritize throughput over latency?**
    - HDFS is designed for read-intensive workloads where sequential reads dominate. High throughput ensures efficient handling of large datasets, while low-latency access is less critical.

3. **Explain the role of blocks in HDFS and why their default size is 128 MB.**
    - The blocks help to distributed large files across the nodes in the cluster, and with large block we thruput in the name node is reduced and minimizes seek time relative to read time.

4. **How does HDFS achieves fault tolerance?**
    - Through data replication across multiple DataNodes, ensuring availability even if nodes fail.

5. **What is the role of the NameNode in HDFS, and how does it manage metadata?**
    - The NameNode stores and manages metadata (e.g., directory tree and block locations) in RAM, enabling fast access. It logs changes synchronously to a journal for consistency.

6. **Compare the roles of the DataNode and Secondary NameNode.**
    - DataNode: Store the actual data and periodically updates the NameNode about the block status.
    - Secondary NameNode: Receives copies of the edit log from NameNode. When the primary NameNode is down, the system uses it and stays read-only.

7. **What happens during a file read operation in HDFS?**
    - The client request for the block location from NameNode
    - Data is read from the closest available DataNodes, often within same rack or node.

8. **Describe the file write process in HDFS.**
    - Client request DataNode for replication, writes data in pipeline manner, and ensures replicas are distributed across racks for reliability.

9. **What is the purpose of scheduling in Hadoop, and what are the key scheduling strategies?**
    - Scheduling allocates resources to tasks efficiently, Key strategies are:
        - FIFO: Processing the tasks based on arrival order.
        - Fair: Balance the work load across the jobs.
        - Capacity: Allocate resources to virtual queues with elasticity.

10. **Why does the FIFO scheduler penalize small jobs?**
    - Smaller jobs may wait behind the larger jobs in the queue, which can lead to delay if the large files dominate the queue.

11. **Explain the role of combiners in shuffle phase:**
    - Combiners reduce the amount of data transferred to network during the shuffle phase performing local aggregation before sending data to reducer.

12. **What is the significance of the shuffle and sort phase in Hadoop?**
    - It distributes and organize data from mappers to reducers, ensuring efficient processing in the reduce phase.

13. **How does Hadoop handle task failures?**
    - Tasks retire four time by default and then marked as failure, if it hangs, it gets killed and restarted.

14. **What mechanisms are in place to detect and handle node failures in Hadoop?**
    - The NameNode monitors DataNode heartbeat, if a heartbeat is missing, the node is removed from cluster, the data replication ensures continuity.

15. **How does the fair scheduler differ from the capacity scheduler?**
    - Fair Scheduler balances the resources across active jobs dynamically.
    - Capacity Scheduler pre-allocates resources to queues but allows elasticity if the resources are unused.
16. **What is the role of Zookeeper in handling scheduler failures?**
    - Zookeeper maintains a backup of the scheduler and ensures failover mechanisms to continue operations without disruption.

17. **Why is metadata storage in the NameNode highly efficient?**
    - Metadata is much smaller than the actual data itself.

18. **Why are tasks in Hadoop designed to be independent?**
    - Independence allows tasks to run in parallel, maximizing resources utilization and simplifying failure recovery by restarting the tasks without affecting others.

Apache Spark

1. MapReduce Pros and Cons: MapReduce revolutionized big data processing by enabling parallel processing across unreliable clusters. However, its reliance on disk-based operations for fault tolerance posed significant limitations:
    - Muti-Stage applications: Iterative tasks in machine learning and graph processing requires repeated data access, which incurs delays with disk I/O.
    - Interactive queries: ad-hoc analyses demand real-time responsiveness, which MapReduce struggles to provide.
2. The Innovation In-Memory processing: Spark's in-memory data handing addresses these shortcomings, allowing iterative and interactive tasks to execute with speedups of 10x to 100x compared to disk-based approaches. This improvement ensures that:
    - Intermediate Data: Can be retain from RAM, reducing reliance on slow disk writes and reads.
    - Fault Tolerance: is maintained without sacrificing performance, thanks to resilient mechanisms like lineage tracking.
3. Resilient Distributed Dataset (RDDs): RDDs form the core abstraction in Spark, designed to effectively handle large-scale data processing:
    - Properties:
        - Immutability: Data remains unchanged once created.
        - Partitioning: Datasets are distributed across cluster nodes for parallelism.
    - Operations:
        - Coarse-Grained Transformations: Functional programming constructs like map, filter and join operate on entire datasets.
        - Fault recovery: Lineage logging enables Spark to reconstruct lost partitions by replaying transformations without additional cost if no failures occur.
4. Cause study (Log mining): Spark simplifies log analysis by leveraging its functional APIs:
    1) Dataset loading: Use textFile to load log data from distribution file system (HDFS).
    2) Filtering: Apply filter to identify error logs.

3) Transformation: Extract relevant information using map.
4) Persistence: Store intermediate results in memory with persist.
5) Analytics: Perform further filtering and aggregations efficiently.

This modular pipeline highlights Spark efficiency in data exploration and iterative queries.

5. Fault recovery via lineage: Lineage tracking is Spark's mechanism for fault tolerance. By recording all operations applied to an RDD:
    - Lost data can be recomputed from the original sources and transformations.
    - Fault tolerance is achieved with minimal overhead, avoiding full replication.
6. Stage and Task:
    - A stage is a collection of tasks that share the same shuffle dependency, meaning they must exchange data with one another during the execution.
    - Shuffle dependency shows a relationship between two stages where data needs to be Shuffled between nodes to satisfy the requirement of next stage
    - When a spark job is submitted it breaks down into several stages based on the operations defined in the code.
    - Each stage is composed of one or more tasks that can be executed parallel across multiple nodes in a cluster. Stages are executed sequentially, with the output of one stage becoming the input of next stage.
7. Narrow/Wide stages:
    - Narrow stages are stages where the data doesn't need to be shuffled. Each task in the narrow stage operates on a subset of partitions of its parents RDD. Narrow stages executed in parallel and can be pipelined.
    - Wide stages are stages where the data needs to be shuffled across the nodes in the cluster. This is because is task in a wide stage operates on all partitions of its parent RDD. Wide stages involved a shuffle and typically are more expensive than narrow stages.
8. Conclusion: Apache Spark, through its focus on in-memory processing and resilient distributed datasets, provides a versatile and high-performance alternative to traditional MapReduce frameworks. Its design principles cater to modern big data challenges, such as iterative computation and real-time query handling, making it a cornerstone of contemporary distributed computing.

Sample questions

1. What is the significance of immutability in RDDs?

    Immutability ensures that RDDs cannot be modified after creation, which simplifies fault tolerance by allowing Spark to use Lineage (a record of transformation) to recompute the lost data. It also enables deterministic computations and thread safety in distributed environments.

2. How does the Spark recover lost partitions in RDD?
    Spark tracks the lineage of transformations applied to an RDD. If a partition is lost, Spark recomputes it by replaying the sequence of transformations starting from original data source.
3. Compare RDD transformations and actions. Provide examples.

Transformations (e.g., map, filter, join) are operations that define how to derive one RDD from another. They are lazily evaluated. Actions (e.g., count, collect, saveAsTextFile) trigger computation and return results to the driver or write data to external storage.

4. Explain how coarse-grained transformations in RDDs differ from fine-grained updates in traditional databases.
    - Coarse-grained transformations apply operations to entire datasets, optimizing parallelism and fault recovery. Fine-grained updates involve modifying individual records, which are more suited for transactional systems like databases
5. Why does Spark perform significantly faster than MapReduce for iterative computations?
    - Spark keeps intermediate data in memory, avoiding repeated disk I/O during iterative computations, such as in machine learning and graph processing. This reduces latency and significantly improves performance.
6. What challenges arise in implementing fault tolerance with in-memory processing, and how does Spark address them?
    - In-memory data is volatile and can be lost if a node fails. Spark addresses this by tracking lineage, enabling it to recompute lost partitions without requiring full replication.
7. Discuss the trade-offs between in-memory processing and disk-based fault tolerance.
    - In-memory processing provides high performance but requires significant RAM and careful fault tolerance mechanisms like lineage. Disk-based processing is slower but more robust as data persists across failures.
8. Identify and explain two major limitations of MapReduce in handling multi-stage applications.
    - Disk I/O Overhead: Each MapReduce job writes intermediate results to disk, causing delays in multi-stage applications.
    - **Lack of Iterative Processing Support**: Repeated disk reads and writes make iterative computations (e.g., machine learning algorithms) inefficient.
9. How does MapReduce handle fault tolerance, and why is it less efficient than Spark?
    - MapReduce achieves fault tolerance by writing all intermediate results to disk and replicating them. This approach ensures reliability but significantly increases I/O costs, unlike Spark's in-memory processing with lineage tracking.
10. What is the advantage of pipeline replication during file writes in HDFS?
    - Pipeline replication ensure data reliability by copy blocks across multiple DataNodes sequentially while minimizing bandwidth usage.

GraphX

1. GraphX in context:
    - Spark's graph processing API extends its capabilities to effectively handle graph-based analytics.
    - Key applications include machine learning and network analysis, leveraging the same computational cluster but with specific optimizations for graph workloads.
    - GraphX is lightweight, with its initial version comparing just 2500 lines of code, and its build upon Apache Spark.
2. PageRank Algorithm: Evaluate how important a node is in a graph, often used in ranked web pages be relevance.

- Mechanism: A random walker traversers the graph. At each step:
    - With probability d = 0.85 the walker follows a random outgoing link.
    - With probability 1-d= 0.15, the walker jumps to a random page.
    - The PageRank score for a node represents the probability of the walker landing on that node.
- Single-machine implementation:
    - Iterative computation updates the rank values based on the contribution from neighbors, scaled by their ranks and the number of their outgoing links.

3. Pregel Model and GAS Framework: Simplifies graph computation by allowing users to "think like a vertex" Computations are expressed as messages changed between vertices.
    - GAS (Gather, Apply, Scatter):
        - Gather: Combines incoming messages.
        - Apply: Updates the vertex state based on the combined messages.
        - Scatter: sends the update messages to the neighbors.
4. Graph Partitioning and Optimization:
    - Partitioning by edge:
        - To optimize distributed graph processing, the graph's edge list is partitioned across machines.
        - High-degree nodes may be mirrored across partitions to minimize message-passing overhead.
    - Mirroring: Instead of sending identical messages multiple time, to the same machine, single mirror node aggregates and forwards the messages locally.
    - Inactive nodes: In iterative algorithms, nodes that converge early are marked as inactive. This avoids unnecessary computations and communication, speeding up subsequent iterations.
5. Distributed Dataflow frameworks and Spark's role:
    - Distributed Framework:
        - Use a directed acyclic graph (DAG) of tasks for parallel computations.
        - RDD's provide fault tolerance and efficient memory usage, essential for iterative graph algorithms.
    - Advantages in GraphX:
        - Spark's in-memory computation avoids redundant data movement.
        - RDD lineage tracking allows efficient recovery without high runtime overhead.
        - Co-partitioning RDDs ensures that graph partitions remain efficient.
6. Graph Programming Abstraction:
    - Property Graph as Collections:
        - Graph data structures are represented as collections (e.g., vertices and edges) that Spark can process using its high-level API.
    - Graph Computation with Dataflow Operators:
        - The abstraction aligns with Spark's data processing model, making it compatible with broader Spark workflows.

Sample Questions

1. Explain the primary motivation behind integrating GraphX into Apache Spark. How does it differ from standalone graph processing engines?
   GraphX integrates graph analytics into Spark, allow data scientists to leverage the same computational resources for both graph-based and general data-parallel tasks. Unlike standalone engines, GraphX reuses Spark abstractions (RDD) for efficient in-memory graph processing and fault tolerance.
2. Describe the data structures used in GraphX to represent graphs. How do they ensure compatibility with Spark's RDDs?
   Graphs in GraphX are represented as property graphs consisting of two RDDs: a vertex RDD and an edge RDD. The vertex RDD contains node identifiers and attributes, while the edge RDD stores connections with attributes, enabling graph algorithms to fit within Spark's dataflow model.
3. Derive the iterative formula used in the PageRank algorithm and explain the roles of d and 1−d.
   PageRank score PR(v) of a node v:

$$PR(v) = \frac{1-d}{N} + d \cdot \sum_{u \in \mathrm{In}(v)} \frac{PR(u)}{\deg(u)}$$

Scalability

1. **Scalability vs. Performance:**
   - Scalability refers to how the system's speed increases when more resources are added.
   - Performance indicates the overall speed of the system, independent of resource allocation.
2. PageRank comparison across systems: The performance of various systems running PageRank, a popular graph-processing algorithm, was compared. The key findings include:
   - Systems with distributed architectures typically utilize more cores but still fall short of outperforming optimized single-threaded implementations.
   - The performance of distributed systems is impacted by bottlenecks like I/O, memory access patterns, and overhead from programming frameworks like Java.
3. Advanced Optimization: Hilbert Curve: To address memory access locality and improve efficiency, an advanced optimization was introduced:
   - Hilbert Curve Sorting: This technique reorders edges to enhance locality in memory access, yielding an additional ~2x speedup.

Concurrency in Python

1. Distributed systems are based on client-server and peer to peer architecture in both cases each node simultaneously acts as client and server.
   - In client-server architecture, a server needs to answer to several clients. One possible solution to avoid bottlenecks is to adopt multi-threading algorithms.
2. Multithreading:
   - Multithreaded programs are programs with multiple concurrent execution flows, called threads, that share the same (virtual) memory space.
   - Threads can be used to write parallel programs. Thread operates faster than processes due to the following reasons:

  - Thread creation is much faster.
  - Context switching between threads is much faster.
  - Threads can be terminated easily.
  - Communication between threads via shared memory is much faster.

Apache Spark(practical)

1. Limitations of MapReduce:
   - Programming model:
     - Hard to implement everything as MR program.
     - Multiple MR steps may be needed also for simple operations.
     - Lack of control structures and data types.
   - Efficiency:
     - High communication cost.
     - Frequent writing of output disk
     - Limited exploitation of main memory
   - Real-time processing:
     - A MR job requires to scan all of the input data
     - Stream processing and random access impossible.

2. Need for new programming model
   - MapReduce greatly simplifies big data analysis.
   - But as soon as it got popular, user wanted more:
     - Multi-stage applications (iterative graphs, machine learning)
     - More efficient
     - More interactive ad-hoc queries
     - Both multi-stage and interactive applications requires faster data sharing across parallel jobs.

3. Motivation and opportunity:
   - Motivation: using MapReduce for complex iterative and jobs or multiple jobs on the same data involves lots of disk I/O
   - Opportunity: The cost of main memory decreased. Hence large main memories are available in each server.
   - Solution: keep more data in main memory, enabling data sharing in main memory as a resource of the cluster.

4. Spark VS. MapReduce in iterative jobs:
   - In MapReduce:
     - Intermediate results are written on disk(HDFS) after each iteration.
     - For each subsequent iteration, data is read back from the disk, creating significant I/O overhead.
   - In Spark:
     - Data is kept in memory (RAM) across iterations.
     - By caching in RAM Spark eliminates the need to read and write data to/from disk.

- Advantage:
  - Faster iterative process
  - Data sharing between iteration happens efficiently through iterations.

5. Multiple analysis of the same data:
   - In MapReduce: when multiple queries are performed on the same dataset, MapReduce reads data from memory for each query.
   - In Spark: Data is read only once from HDFS and then stored in RAM. This cashed data is shared among all queries without reloading it from disk, saving time and resources

6. Key features of Spark's data sharing:
   - Spark uses RDD to manage data in memory across transformations.
   - RDDs enables Spark to process data efficiently without intermediate disk writes.
   - Each node in the cluster holds a partition of dataset in its main memory allowing parallel processing and reducing the load on any single node.

7. Spark Data Flow
   - The data initially read from HDFS and this data is served as the input to Spark. However, unlike MapReduce, Spark perform most of the operations in memory instead of writing back on disk after every stage.
   - MapReduce has a strict 2 stage operation. Map and Reduce, Spark allows more flexibility, multiple stage of transformation all happening on memory.
   - Extract the relevant portion of data and allows the data (RDD or data frame) to cached in memory.
   - In Spark you chain multiple transformations (filter, groupby, aggregate) in a single job without requiring intermediate writes on disk after each stage.
8. Data representation as RDD:
   - Everything you can do in Hadoop; you can do in Spark but it relies on Hadoop (HDFS) for storage.
   - Spark computation paradigm is not just a single MapReduce job, but a multi-stage, in-memory dataflow graph based on RDD.
   - Partitioned/Distributed collections of objects spread across nodes of a cluster.
   - Stored on RAM or on local Disk.
9. Spark computing framework.
   - A programming abstraction based on RDD and transparent mechanisms to execute code in parallel on RDD.
   - Manage job scheduling and synchronization.
   - Mange the split of RDD in partitions and allocate RDDs' partitions in the nodes of cluster.

	Hadoop MapReduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and reduce	Map, reduce, join, aggregate
Execute model	batch	Batch, interactive, steaming
Programming Environment	Java	Jave, R, Python

10. Key components of Apache Spark:
    - Spark core: The backbone of spark; handles:
        - Task scheduling: distributed tasks across nodes.
        - Memory management: Efficient usage of in-memory storage.
        - Fault Tolerance: automatically handles failures.
    - Extended libraries:
        - Spark SQL: enables structured data queries using SQL or DataFrame API.
        - Spark Streaming: Process real-time data streams.
        - MlLib: a library for ML and data mining algorithms
        - GraphX: for graph and network analysis.
11. Executing in Spark:
    - Driver program: The user program runs on the driver, which:
        - Coordinates tasks across workers.
        - Schedules and aggregates results.
    - Worker nodes:
        - Executes task using executer.
        - Each node stores data in-memory and performs computation on distributed chunks.

RDD

1. Definition: Is the primary abstraction in spark. They are distributed collections of objects spread across the nodes of cluster.
    - Each node of the cluster that is used to run an application contains at least one partition of the RDD(s) that is(are) defined in application
    - RDDs are stored in main memory of the executer running in the node of the cluster, or on the local disk of nodes if there is not enough main memory.
    - RDDs allow executing in parallel the code invoked on them; each executer of a worker node runs the specified code on its partition of the RDD
2. RDDs and Data Flow Graph
    - RDDs are immutable once constructed: the contents of RDDs cannot be modified.
    - Track lineage information to efficiently recompute the loss data.
    - This information is represented by means of DAG connecting input data and RDDs.
3. RDDs creation: RDDs can be created by:
    - Parallelizing existing collections of the hosting programming languages(collections and list java, python, or R)
    - From large files stored in HDFS or any other file system. There is one partition per HDFS block.
    - By transforming an existing RDD, the number of partitions depend on the type of transformation.
4. Operators on RDDs:
    - Transformation:
        - Create a new dataset from an existing one.
        - Lazy in nature, there are executed only when some action is performed.
        - Exp: Map, Filter, Distinct.

- Actions:
  - Return to the driver program a value or exports data to a storage system after performing a computation.
  - Exp: count (), Reduce (), collect, Take ()
- Persistence:
  - For caching datasets in-memory for feature operations
  - Option to store in disk, RAM, or mixed (storage level).
  - Exp: Persis (), Cache ()

5. Transformation: Two kinds of transformations:
   - Narrow transformations: They are the results of map, filter and such that the data from a single partition, it is self-sustained.
     - An output RDD has partitions with records that originate from a single partition in the parent RDD.
     - Spark groups narrow transformations as a stage.
   - Wide Transformations: They are the result of groupByKey and reduceByKey.
     - The data required to compute the records in a single partition may reside in many partitions of parent RDD.
     - All of the tuples with the same key must end up in the same partition, processed by the same task.
     - Spark must execute RDD shuffle, which transfers data across the cluster and results in a new stage with a new set of partitions.
6. Actions: actions are synchronized operations that return values.
   - Any RDD operation that returns a value of any type but RDD is an action. (Exp: SaveAs, Collect, Take, Reduce.)
   - Actions trigger the execution of RDD transformations to return values
   - Until no action is fired, the data to be processed is not even accessed.
   - Only actions can materialize the entire process with real data.
   - Actions cause data to be returned to driver or saved to output.
   - Main Actions (reduce, collect, count, first, take, takeSample, takeOrder, saveAsTextFile)

7. Persistence: by default, each transformed RDD is recomputed each time you run an action on it, unless you specify the RDD to be cached in memory.
   - RDDs can be persistent on disk as well, caching is the key tool for iterative algorithms.
   - Using persist(), one can specify the storage level for persisting an RDD.
   - Caching allows us to avoid iterative recomputation of the data.

Spark programs

1. Common spark use cases:
   - Spark context and RDD creations:
     - connection and leading data into Spark context.
   - Basics RDDs:

        - Unary RDD transformations & actions
        - Binary RDD transformations & actions
    - Key value pairs:
        - Unary RDD transformations & actions
        - Binary RDD transformations & actions
2. Spark Context: is the basic entry point to the spark runtime environment and underlying file system (HDFS).
    - Once you have Spark Context, you can use it to build RDDs.
3. RDD transformations: transformations take one RDD input and return another RDD as output.
    - They define the logical structure of data flow, but they don't actually trigger any computations
4. RDD actions: Actions take RDD object as an input and perform a final operation to obtain a non-distributed, either mutable or mutable object as output.
    - Actions trigger the actual computation workflow.
5. Persistency and caching of RDDs: The previous dataflow program can access RDD twice actions: a count() and a take().
    - To avoid repetitive computation of the badlinesRDD from the input RDD, we can explicitly persist the former within the cluster's main memory.
    - Just call the persist() on the RDD before the actions are invoked.
    - If main memory is full, spark uses a LRU policy to free cached RDDs.

Spark SQL

1. Introduction: make data look like table regardless how they lay out. Generate a specific execution plan for this query.
2. SparkSQL Vs. Spark:
    - No data parsing
    - Syntax is easier
    - Code optimization
    - No overhead (overheard refers to additional computational and programming effort require to process data.
            - Effort in writing and maintaining code + inefficient query execution.
3. Spark as a database language? Application scenario
    - No real-time queries (High latency): Spark process the data in batches, so depending on the dataset size, it might take too long to process and answer a query.
    - No row level updates: since Spark uses RDDs to store data, after creating the database, we can't alter or update it. The only way is to rewrite the whole partition of data.
    - Not designed for online transaction processing: Spark SQL is designed for online analytical processing (OLAP) which focuses on complex queries on large datasets. But online transaction processing (OLTP) involves frequent insert updates, deletes and small transactional queries which spark can't handle well.
    - Best use: Batch jobs over large sets of append-only data (such as logs, event streams or sensor data)
        - Log processing
        - Data/Text mining

    - Business intelligence

4. What's Hive? A big data management system storing structured data on Hadoop file system.
    - It projects tabular schemas over folder and files in HDFS
    - Enables the content of folders in HDFS to be queries and tables, using SQL-like query mechanism.
5. Structured API:
    - The interfaces provide more information about the structure of both the data and computation being perform.
    - Spark uses this extra information to perform extra optimizations based on SQL-like optimizer called Catalyst.
        - Compute the best execution plan before executing the code.
        - Programs are usually faster than standard RDD-based programs

6. DataFrames: RDD + schema
    - Schema awareness: data frames are schema-aware. This means the system understand the structure of data (column name, data types) and can apply optimizers.
    - A distributed collection of rows organized into name columns.
    - An abstraction of selection, filtering aggregation and plotting structured data.
7. Methods:
    - Select: choose specific column
    - Filter: filter rows based on a condition
    - GroupBy: group data by a specific column
        - agg: perform function like (count, sum, max, min average, ….)
    - join: combine multiple data frames based on specific column (inner, left, right, cross, out)
    - sort/orderBy: arrange rows based on one or more column
    - drop: remove a specific column
    - distinct: get unique rows
    - Alias: rename column

Sample questions

Dataset 1: l1.csv -> user_id, date, track, artist, album   Dataset2: g1.csv -> artist, genre

1. Upload the csv file into a dataframe:

    ListenDF = spark.read.csv("listen path"), genreDF = spark.read.csv("genre path")

2. The 20 most listened songs:

    Group by track -> agg(counts)
    ListenDF.select("track").groupby("track").agg(count(*)).sort("count",desc()).show(20)
    # or
    Tracklist = ListenDF.groupby("track").count()
    Tracklist.sort(desc("count")).show(20)

3. Top 20 most listened genres:

    DF3 = ListenDF.join(GenreDF, on='artist')
    groupbyGenre -> DF3.groupby('genre').count()

```
groupbyGenre.sort(desc).show(20)
# or
Genres = genreDF.join(ListenDF, “artist”).select(“genre”)
GroupGenres = Genres.groupby(“genre”).count()
GroupGenres.sort(desc(“count”).show(20)
```

4. In year 2015 for each artist and month, the number of listened songs:

```
Listened2015 = ListenDF.select(*).where(year(“date”)==2015)
Listend2 = Listened2015.groupby(month(“date”)).groupby(“artist”).count()
Listend2.show()
```

5. Dataset3: network -> user_id1, user_id2
6. Friends at Dataset3 without repartitioning:

```
Consider user = “Bob”
networkDF = spark.read.csv(“network.csv”)
Bob_net = networkDF.select(*).where(user_id1 = “Bob”).distinct()
Dist_Two = Bob_net.join(networkDF, on=Bob_net.user_id2 == networkDF.user_id1)
Dist_three = Bob_net.join(networkDF, on=Dist_Two.user_id2 == networkDF.user_id1)
Dist_three.distinct().show()

# or

For I in range(3):

    new = networkDF.join(all, “user_id1”).select(“user_id2”).distinct()

    all = all.union(new)

    all.show()
```

Stream Spark

1. Introduction: Spark streaming provides a high-level abstraction called discretized stream or DStream, which represents a continues stream of data.
   - DStream can be created either from input data streams from sources such as Kafka, or by applying high-level operations on other DStreams.
   - Internally, DStream is represented as a sequence of RDDs.
2. Socket DStreams:

```
sc = SparkContext(“local[2]”, “NetworkWordCount”)
ssc = StreamingContext(sc, 1)
```

   - Using this context we can create a DStream that represents the streaming data from a TCP source. Specified as hostname(local), and port(999)

```
Lines = ssc.socketTextStream(‘local host’, 999)
```

- The lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a lone of text. Next, we want to split the lines by space into words.

  ```
  words = lines.flatMap(lambda: line.split(" "))
  ```

- Flat map is one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source stream. In this case, each line will be split into multiple words, and stream of words is represented as the words DStream, next we want to count these three words:

  ```
  Pairs = words.map(lambda word: (word,1))
  Wordcounts = pairs.reduceByKey(lambda x, y: x+y)
  Wordcounts.pprint()
  ```

3. Producer and Consumer: When these lines are executed, Spark streaming only sets up. The computation when its started, and no real processing has started yet.
   - To start the processing after all the transformations have been setup, we finally call:

     ```
     ssc.start()
     ssc.awaitTermination()
     ```
4. Transform: Return a new DStream by applying RDD-to-RDD function to every RDD source DStream

Structure Streaming

1. Definition: Structure Streaming is a fault-tolerance and scalable streaming processing engine built on the Spark SQL engine.
   - The Spark SQL engine takes care of the running program incrementally and continuously and update the final result as a streaming data continues arriving.
   - It is possible to use DataFrame instead of RDDs.

Sample Questions

Base: Getting stream of data from a producer and doing "word counting" in each bactch.

1. Word counting in each batch:
   ```
   Lines = ssc.SocketTextStream("localhost", 999)
   Word_counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word:
   (word,1)).reduceByKey(lambda a,b: a+b))
   Word_counts.pprint()
   Ssc.start()
   Ssc.awaitTermination()
   ```
2. Count the number of lines in each batch:
   ```
   Lines = ssc.SocketTextStream("localhost", 999)
   Lines.count().map(lambda x: lines in this bach: %s %x).pprint()
   # or
   Rdd = line.count().pprint()
   Ssc.start()
   Ssc.awaitTermination()
   ```

3. Word_counting by windows:

   Ssc.checkpoint(“hdfs:/user/…path)

   Lines = ssc.SocketTextStream(“localhost”, 999)

   Lines.countByWindow(10,5).map(lambda x: lines in this window %s’%x).pprint()

   Ssc.start()

   Ssc.awaitTermination()

4. Create a new RDD with “sentiment” weights” associated to each word:
   Index = ssc.parallelize[(‘smile’,0.8),(‘sad’,0.1),(‘funny’,0.8), …, (‘dog’, 0.5)]
   # create a static RDD called index to store word + sentiment weights
   Lines = ssc.SocketTextStream(“localhost”,999)
   Word_count = lines.flatmap(split(“ “)).map(lambda x: (x, 1)).ReduceByKey(lambda x,y: x+y)
   Happiest_words = word_counts.Transform(lambda rdd: index.join(rdd))\
   .map(lambda x: (x[0], round(Float(x[1][0]) * x[1][1], 2)))\ # sent_score x count (rounded)
   .map(lambda w: (w[1],w[0])\ # reverse the key-value pair to sort by score
   .transform(lambda rdd: rdd.sortByKey(False)) # sort the words by their sentiment score (descending)
   Happiest_words.ForEachRDD(print_words)

PySpark MLlip Pipelines

1. Main Concepts in Pipelines: MLlib standardizes APIs for ML algorithms to make easier combination of multiple algorithms into a single pipeline, or workflow. In includes:
   - DataFrames: DataFrame from Spark SQL as ML datasets. It contains different columns storing text, feature vectors, true labels, and predictions
   - Transformer: It’s an algorithms which can transform one DataFrame into another DataFrame. A ML model is a transformer which transforms a DataFrame with features into a DataFrame with predictions.
   - Estimator: is an algorithm which can be fit on a DataFrame, to produce a Transformer, e learning algorithm is an estimator which trains on a DataFrame and produces a model.
   - Pipeline: A pipeline chains multiple transformers and estimators together to specify an ML workflow.
   - Parameter: All transformers and estimators now share a common API for specifying parameters.
2. Transformers: It implements a method transform (), which converts a DataFrame into another, generally by appending one or many columns.
   - A Feature Transformer might take a DataFrame, read a column map it into a new column (feature vector) and output a new DataFrame with the mapped column append.
   - A Learning Model might take a DataFrame, read the column containing the feature vector, predict the label for each feature vector, output a new DataFrame with predicted labels appended as new column.
3. Estimator: Abstract the concept of a learning algorithm that fits or train on the data.

- An Estimator implements a method fit (), which accepts a DataFrame and produces a model, which is a transformer.

4. Pipeline: is a specified sequence of stages and each stage is either a Transformer or an Estimator. These stages run in order, and the input DataFrame is transformed as it passes through each stage.
    - For estimator stages, the fit() method is called to produce a Transformer (which becomes part of the pipelineModel), and the Transformer's transform() method is called on the DataFrame.

5. Pipeline Example:
    - Raw Text will be the input of our Tokenizer (transformer) and returns words (feature vector)
    - The words are the output are given to HashingTF(transformer) and returns numerical feature vectors.
    - The new DataFrame (numerical feature) will be the input of Logistic Regression (Estimator) and the output will be the model
6. DAG pipelines: A pipeline's stages are specified as an ordered array
    - It is possible to create non-linear Pipelines as long as the flow graph forms a Directed Acyclic Graph (DAG).
    - This graph is implicitly based on the input and output columns names of each stage (generally specified as parameters).
    - If the pipeline forms a DAG, then the stages must be specified in the topological order.
    - A pipeline stages should be unique instances. However, different instances can be put into the same pipeline since different instances will be created with different IDs.

1. What are the trade-offs between ACID and BASE model in distributed systems?
    - ACID (Atomicity, Consistency, Isolation, Durability) prioritize strong consistency making it ideal for banking, but leading to lower availability.
    - BASE (basically available, Soft state, Eventual consistency) prioritizes availability and partition tolerance at the cost of strong consistency, making it suitable for large scale distributed systems (DynamoDB).
    - Trade-Off: ACID ensures correctness but at the cost of a rigid and slower system in large scale. BASE provides flexibility and high-tolerance at the cost of consistency and stale data.
2. How does the CAP theorem impact the design of distributed systems?
    - The CAP theorem states that in a distributed system it only can have only two of the three of below attributes:
        - Consistency: all nodes see the latest data at the same time.
        - Availability: Every request receives a response (even if stale).
        - Partition Tolerance: The distributed system be functional if some nodes or network fail.

- Impacts:
    - Systems like google spanner priratize consistency over availability (CP).
    - Systems like Dynamo prioritize availability over consistency (AP)
    - Systems like Zookeepr ensures both consistency and partition tolerance but may sacrifice availability

3. How does eventual consistency work and it's different from strong consistency?
    - Eventual consistency guaranties that all replicas in the system will reach consistency over time and converge if there is no new update
    - Strong consistency ensures that all nodes in the cluster will have the latest update immediately. It's used in traditional data bases like google spanner.
4. What problem does the Paxos algorithm solve in distributed systems?
    - Paxos is a consensus algorithm