**General**

**Distributed System**

It is an application that executes multiple protocols over a network of machines to achieve some goals.

**Why do we need distributed systems?**

* Systems are now inherently distributed. ( Sending and receiving data over the network )
* Better reliability ( fault tolerance )
* Better Performance (improved query (specially read) throughput through replication)
* Solves the single machine problems like ( limited storage capabilities, limited GFLPOs, geolocality)

**Challenges of a distributed system**

* Fault tolerance. How to recover from failures without performing incorrect actions
* Availability of 24x7 operations, even in the presence of failures!
* Recovery of failed system component can restart and rejoin the system in a correct state.
* Consistency or correctness System invariants are preserved in presence of concurrency, asynchrony, and failures.
* Scalability, being able to handle large amount of user’s requests and able to store large amount of data that can’t fit on a single machine.
* Performance achieves predictable performance, response in timely manner.
* The security system has secured its data and code.

**Interface Definition Language (IDL)**

provides language-independent type signatures of the functions that are being made available over RPC.

For gRPC, they use protocol buffer that uses the .proto files to specify the schema of the messages request and response and the compiler will generate the serialization and deserialization code, to make the calls using gRPC.

**MapReduce:**

It is part of Apache Hadoop ecosystem as distributed data processing unit, that processes large amount of data on a cluster of machines. MapReduce runtime library takes care of parallelization, synchronization, fault tolerance, communication, load balancing and scheduling.

And it is used by creating Map and Reduce Function. Map(Key, value) := tuple in a Db. And output list of keys and values. The reduce function after a shuffle and sort phase in the network. It will take a key and list of all its values and then we can do something about it.

**MapReduce Architecture:**

It is a master slave architecture. That consists of the followings:

**Master node:**

* **Name Node (HDFS):** has the meta data of where the files are stored and how they got chunked using 64 MB (default) chunks and stored on different data node with replication(default 3) for fault tolerance. They use big chunks to reduce the number of metadata being stored on the master node to keep it light weight.
* **Job Tracker (MapReduce):** it is responsible for fault tolerance and when a MapReduce job is submitted it will assign the tasks to different Task Trackers with the help of YARN(Yet Another Resource Negotiator)

**Worker nodes:**

* **Data Node (HDFS):** this is where we store the data during the shuffle and sort phase, node communicate and get data using HDFS.
* **Task Tracker (MapReduce):** it is responsible for the job assigned by the job tracker locally on its machine, it must send some kind of heartbeat messages to the job tracker to ensure liveliness. If the master suspected any failed node. It will assign its job to a different free slave node and the data is not lost due to saving all the intermediate step in HDFS.

**MapReduce Designs Principles:**

MapReduce operates on large datasets that can’t fit on a single machine and that the job runs on the cluster of commodity machines that may crash at any time.

**Leases:**

It is a scalable resource management in distributed systems. **Definition:** A lease is a contract that gives its holder specified rights over a resource for a limited period.

Normally when multiple processes share a resource, each process gains exclusive access (write) to the resource by acquiring a lock on the resource and then releasing the lock when it finishes.

**Problem:** is that the process can crash so the resource will be locked indefinitely.

**Solution:** use a lease (fault tolerance alternative) to gain access to the resource and each lease has an expiration time. When it expires the resource will be free for other processes.

**Types:**

**Read Lease (Granted concurrently)** **:** the process can get a read access to a resource, that allows it to cache the data and guarantee that no other process is modifying the data.

**Write lease (Granted exclusively) :** the process will be able to modify the data locally and write it in batches. It guarantees that no other process has the data cached. Only the changes will be sent to the server to avoid sending large data using offset and length.

**How to use it:**

Client\_Request(res\_id, READ | WRITE) with the type of the lease.

The Server (leases grantor) can send an eviction notice to the process to reclaim the resource. So, the read leases must clean their cache and the write leases must write back.

**Fault Tolerance:** in case the lease holder crashes, the server will wait for the expiration time and reclaim the resource. But if the server crashes, on recovery it will wait for the clock skew and the lease expiration time to reclaim the resources.

**Clock Skew between the server and the lease holder might lead to inconsistencies because of the expiration date. How to solve this issue:**

It can be solved by using NTP to reduce the clock skew of both using a more accurate source of time. But this also might lead to some inconsistency due to the different latencies because of the geolocality of the lease holder and generator. So, we can synchronize their clock w.r.t each other using NTP. That would reduce the clock skew between them and leads to a better understanding of the time left for the expiration of the lease.

**Apache Pig:**

Motivation because of the limitation of MapReduce, of there is no implementation for primitive data analytics tasks like joins, group by, sorting and because the programmers are lazy to write map and reduce. So, they abstracted all the primitive operations to avoid rewriting primitive operations.

It is built on top of MapReduce. It has pig Latin: **SQL query like + distributed execution.**

So, the pig compiler will first generate a logical plan to optimize the query if it is optimizable and then it creates the physical plan and then create MapReduce pipelines.

**Advantages of staged compiling:**

Query optimization and MapReduce specific Optimization (like the global arrangements is given by the MapReduce shuffle and sort phase from the runtime library).

**Google File System:**

It is like HDFS for Google. We need to avoid the limitation of a single machine like the bandwidth and the storage. So, using cluster of machines to aggregate the space and the bandwidth to handle the scalable data storage. So normal file systems on a single machines were not able to handle large amount of data of Google due to its limited memory.

Google created GFS to get access to huge amount of data in a quick manner. They are sharding of the files into hundreds of machines and use replications for fault tolerance aiming for automatic recovery of nodes. And they aim to gain fast access by big sequential reads.

**What is a Distributed File System:**

It is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing to access files from any network or computer.

**GFS Architecture:**

* **Master node:**

It stores the metadata of the files which is mapping between the file names to where to find the data. It divides the files into chunks of 64MB to reduce the number of metadata stored in the master. And it stores a list of the chunks where each chunk is stored (on which chunk node).

**Data on the master:**

* + filename -> [chunks handle Ids] (Disk)
  + each chunk handle -> list of chunk servers (replicas) (Memory)
    - -> version for each chunk (Disk)
    - -> chuck is the primary or not (Memory)
    - -> expiration of lease to be the primary (Memory)

stores everything on the disk and memory for fault tolerance and it keeps Log for any change. (Disk)

* **Chunk node:**

Stores the actual data.

**Read Operation:**

The client will contact the master giving an offset and a length and the master will provide where the data is located, and the client will send a read request to that specific chunk server or the closest server to the client given the data will be replicated on multiple machines. Then the chunk returns the data, each chunk is saved in a file.

**Write Operation:**

Record Appends: clients want to append to a file an array of bytes. It talks to the master to ask about which chunk server to talk to that has the last chunk of the file to append to it.

For each chunk there will be primary and secondaries replicas that store the data for crash tolerance. So, GFS uses Replications to fix the crashes.

**If no primary:**

Master must find the up-to-date primary using the version of the chuck server. So, when they all answer it will assign a primary and secondaries to replicate the changes afterwards, so they are using state machine replications. And send the client who are the primary and secondaries. All updates happened using 2PC protocol (if atomic) where all the secondaries have appended the bytes. The replication will happen using state machine replications.

**The GFS architecture is designed to tolerate two types of faults: crash faults and arbitrary-state corruption. Briefly specify How these faults are anticipated and how they are tolerated.**

**Crash faults** occur when a node crashes, to anticipate it can use failure detectors when send a message to a chunk server and there is response within a timeout then the chunk serve might be down, so, to tolerate this, GFS implements a replication strategy (primaries and secondaries using state machine replication) in which all data is stored on multiple nodes. This ensures that, in the event of a node failure, the data is still accessible on other nodes.

**Arbitrary-state corruption** occurs when data on a node is modified in an unexpected or malicious way. To anticipate and tolerate this type of fault, GFS implements a check-summing and versioning strategy in which all data is check-summed and any modifications to the data are stored as new versions. This ensures that, if data is corrupted, the original version of the data can still be recovered.

**Protocol: for update interface**

* The client sends a request to the GFS Master to perform an atomic update on the specified file, including the offset, length, and data to be written.
* GFS Master responds to the client with an acknowledgment.
* The client sends a message to the GFS Chunk servers containing the same information as before.
* GFS Chunk servers reply to the client with an acknowledgment of the request.
* The client sends a message to the GFS Master informing it that the chunk servers have acknowledged the request.
* GFS Master sends a message to all the GFS chunk servers to begin the atomic update process (commit). Like in 2PC so the masterworks as a coordinator node.
* Each GFS chunk server locks its chunk and updates the data. In case of any error, an error message must be returned to the master.
* Once all the GFS chunk servers have written the data atomically, the GFS Master sends a message to the client informing it that the atomic update process was successful.

**Monolithic Kernels**

All OS services operate in kernel space, it has good performance like Unix, Linux kernels.

**Disadvantages**

* Dependencies between system component
* Complex & huge (millions(!) of lines of code)
* Larger size makes it hard to maintain.

**Micro Kernels**

Minimalist approach, add minimal system operation in the kernel space like IPC, virtual memory, thread scheduling and put the rest into user space like device drivers, networking, file system and user interface. It is more stable with less services in kernel space. Example: AmigaOS, Minix

**Disadvantages**

* Lots of system calls and context switches

**Zookeeper**

A coordination (micro-)kernel uses minimalistic APIs that can be used to build a wide range of coordination primitives. storing metadata.

**Zookeeper design principles:**

* FIFO ordering for clients
* Linearizable writers which means being able to order the concurrent writes.
* Wait-free APIs (so the clients won’t get blocked)

It has data structure like a tree stored in memory and disk in case of failure. And the only way to update is to use the Zookeeper APIs. These data are stored in znodes in tree data structure using path. Recommendation is to store small data and not big data files. Because the znodes are stored in memory and this way we will use a lot of memory. and we access the znode using its path.

**Guarantees**

* Client requests are processed in FIFO order.
* Writes to Zookeeper are linearizable.
* Clients receive notifications of changes before the changed data becomes visible.

**Zookeeper APIs**

* create(path, data, flags) ex: create(“/app/resources/res\_1/locks”, NULL, EPHERMALS | SEQUENTIAL)
* delete(path, version)
* exists(path, watch)
* getData(path, watch)
* setData(path, data, version)
* getChildren(path, watch)
* Sync()

Watch flag will only notify the client that something changes so we need to getData again after a notification happened on the znode.

Using zookeeper to solve some kind of synchronization problem between multiple nodes.

**ZAP**

Zookeeper uses primary backup replication model. Where the primary do the changes to its state and propagate the changes to the slaves. All changes are idempotent changes allowing duplicated changes which won’t affect the result. And it uses at least once semantics. Zookeeps uses FIFO for all clients requests so also all the slaves have to execute the operation in FIFO way where all the changes from the leader(primary) will be executed in the same order that the leader broadcast it.

It uses atomic broadcast protocol where all the followers must execute the changes for the operation to be successful.

**Reads:**

Reads happen locally from each machine locally, no need for communication between the nodes.

**Ensuring Linerizable writes works with Quorum.**

All transactions that are the operation that changes the state inside the node are identified by **zxid.** zxid consists of an epoch number and counter, whenever there is a new transaction, the counter is incremented and whenever there is a new leader the epoch is incremented. Using these two variables Zookeeper can ensure linearizability of writes. So, if I have two transaction t1 and t2 then we say that t1 happened before t2 iff:

T1.epoch < t2.epoch or (t1.epoch = t2.epoch and t1.counter < t2.counter)

And this helps in ensuring that the transaction are executed in the same order coming from the leader in FIFO order. So the counter act like a Lamport clock for the events order.

**Failure Detector:**

For a leader he has to receive heartbeat from a quorum if he didn’t he will step down to follower and invoke the leader election to phase one.

For a follower if he is not receiving heartbeat from the leader for a timeout then it will trigger a leader election to phase one which is the Discovery Phase.

**The Leader election:**

* **Discovery Phase**

All followers will send the latest epoch number they have to the candidate and then upon receiving the epochs from a Q, it will propose a new epoch by incrementing the highest epoch the leader receives and broadcast it to the other followers. Followers will update their epoch and send ACK with the highest transaction id they have zxid. The leader upon receiving all the highest transaction ids from the followers receives all missing transactions he does not have.

* **Synchronization Phase:**

The leader proposed himself as the new leader by sending the highest zxid that he got from the quorum. Followers will accept him as leader and receive all the missing transactions that they do not have and send an ack to the leader. Upon receiving the acks from a quorum, the Leader will send a commit request to all followers and followers will deliver all the transactions they have.

* **Broadcast Phase:**

Leader proposes new transactions after receiving a write request and changing it an idempotent transaction and increment the counter and broadcast it to all the followers. After receiving an acks from the follower the leader will deliver the change and send a commit request to the followers to deliver the new txs.

**Key-Value Stores (HBase)**

Highly distributed nodes that store multiple containers (NoSQL) store any complex data type in key value format. They offer higher availability and more scalability. They are more flexible and faster since there is no schema with no relation between the different objects. It gets accessed with simple APIs like **get(key) and put(key, data).**

**Why do we need them?**

* Horizontal scalability: adding and removing nodes to store and support high demands.
* Performance: improves the query throughput and runtime from any replica including geolocality.
* Reliability: replications
* Availability in case of any network partition

**Example:**

* **BigTable**
* **Apache HBase and Casandra**
* **AWS Dynamo**

**Consistence Hashing:**

Used to hash the keys to distribute the data among multiple machines. Map different keys to different nodes. We can use multiple nodes to cache the data coming from the server. And that improves the query runtime by not letting all the requests go to the server but going to one of the cache servers but in case of a miss then it will contact the server and get the data from the server.

We also need to have an even distribution among all the caches we have. And its case of any failure we don’t need too much cache misses. And we need to distribute the data evenly and avoid any bottleneck. We can use:

* **Hash functions**

Hash Function(key) = key % numberOfNodes

Given the key to the data we can know the location on which node that data is located. But in case of removing a node from the system will lead that we need to redistribute the data again among the nodes. And in case of caches, we will have a lot of cache misses and that leads to fetching data from the server.

* **Distributed hash table:** the idea is to avoid all these misses, so we hash the data and the nodes into a ring structure. Where each data or node is given a unique m-bit id and the circle has 2^m bucket. And then each data point will be stored in the next node after it in the ring clockwise direction. So each node’s key space is between the key of the previous key + 1 to its key.

To avoid hash collisions by having higher number of buckets to the hash functions or using a better hash function like SHA256 with no collisions. **MD5 creates 128 bits (16 bytes) hash values.**

So adding and removing nodes to the system will minimize the data movement or distribution of the data. Because the data that was stored in the removed node will be stored and moved to the next node in the ring.

At each node we can speed up the lookup operation by storing the data in a binary tree depending on the key and then the lookup will become O(log n)

**Cassandra Base on AWS Dynamo**

Eventual Consistency and it uses consistent hashing and gossip protocol to share the metadata in the network.

So, inside each node in the system, they use WAL (write ahead log) for failure recovery so any operation will be written log(disk first) so it can reconstruct its state in case of any failure. And the data is stored in LSM trees that consists of Memtable to cache the data in the memory and SSTable for flushing the data to the disk.

All nodes can use some kind of Finger Table for faster O(log n) look up where is the key in the network. So looking for any key it will start with random node. Firstly, it will check it locally. Or it will check the distributed hash table to know which nodes to talk to.

**Read Operation**

Client contacts one of the nodes, then that node becomes the coordinator of the operation, it checks first if it has the data that client is looking for, it not, the using the distributed hash table it can locate the node that is responsible for the key and get the data from that replica and return it to the client.

**Write operation**, the client sends a write request to a node, and it becomes the coordinator and using consistent hashing to determine the node that will store the value and replicated to the other two nodes after the original node. And then the coordinator will send the value and it will response to the client if at least a quorum added the value (by receiving acks from it) to reduce the latency for the client.

**In the case of scaling out adding more nodes** to the networks ( adding more resources to support higher number of requests) takes a random location and split the key space of the following node and the change will be gossiped in the system between nodes to update their hash table.

**In case of failure**, it has a heartbeat mechanism to detect is any node has failed, then will it gossip the info to the others to modify the data. And the data will be replicated on the next two nodes of the failed one. So, no data movement will be needed.

**Log Structured Merge Tree(LSM tree)**

It is the data structure of BigTable and Spanners, this is how they store the data. Especially in NoSQL systems, they manage the data in memory for fast look ups and in disk for durability in case of failures.

**Why do we need this data structure?**

We need it to overcome the limitations of the main memory and disk at the same time. The main memory has fast random reads and write but it is not persistent, and the disk has a slow random reads because of the seek time but the data is persistent. So LSM tree merge the two type of memory into one data structure to avoid their limitations.

**How does it overcome the limitations?**

It is a hierarchical, sequential disk-oriented data structure. It provides random reads from the memory and when it gets data from the disk it accesses them sequentially which is faster than the random reads.

**Simple API**

* Put (key, data)
* Get (key)

**It consists of levels**; the first level is kept in memory and the others on disk as we go down the level the file sizes get bigger. It also uses **WAL (write ahead log)** for providing failure tolerance. It has an append-only interface **(REDO and UNDO)** like a queue where you add to the tail and read the head. And once a log entry is processed then it will be garbage collected.

**It’s data structures:**

In memory: **MemTable (mutable)** using skip list look up O(log n)

In Disk : **SSTable (files immutable)** can’t modify it, only read. It is key value pairs; also, binary search and it maintain it because it is immutable.

**Write Operation:**

Given a new key value pair, the operation will first be appended to WAL for failure tolerance in case of any crash and the key will be added to the Memtable in memory. Once the mem table is full and reached its configurable size it will be marked as immutable and during the compaction phase it will be flushed to the disk as SSTable file, and it will become the new first level of the SSTable. (During compaction there will be garbage collection for the keys that have new values so the old values will be deleted to reduce the number of levels and thus free up some memory.

**Read Operation:**

The key will be looked in the Memtable in the memory and returned if it was found, if not the search will go to the disk level by level and look for it sequentially in the levels using binary search of O(log n)

**Update Operation:**

Add new key value with the same key and during the compaction phase the old value will be garbage collected. And because we search in serial order. So, the most recent value will be returned either ways.

**Delete Operation:**

Add a new entry with same key and value is tombstone like NULL and it will be deleted using the garbage collection.

**BigTable (HBase is the open-source version from Apache)**

It is a distributed multi versioned storage system for structured data. It is sparse so not all data fields are populated. It stores the data in rows and there are column families, and each family has different columns.

**Key(row, column, time) -> value**

**Motivation**

It is built on top of GFS. GFS was limited to unstructured data and the data that google seeks most of it is structured data that has some kind of schema like the web data and per user data so GFS has no understanding of structured data.

MySQL and these kind of companies, the scale broke them that’s why Google developed BigTable to handle large amount of structured data.

**Scale up: get better machines, better performance and more computational powers limited to the expenses.**

**Scale out: adding more machines to create a supercomputer.**

**Data Model:**

It stores the data in a table consisting of row and columns, each row is identified with a key which are sorted in lexicographical order and the columns are organized in column families. And values are stored with timestamps so we can go back in time to check older values.

Values are accessed like (ColumnFamily:columnName) they use column families to have different access right to different families. (Access Control). Additionally, it stores the data in a columnar style and the columns are compressed together.

Tablets: Collection of rows.

Example: Google crawlers, stores data where the URL is the key, and the values are the content of the web page over time.

**Important**

BigTable does not support transaction on multiple rows, it supports only single row transactions (so it guarantees the atomicity on a single row only which is a big limitation)

**Timestamp:**

* Versioning: used to store different versions of data in a cell
  + 64-bits integer (UNIX timestamp)
  + New writes default to current time, but timestamps for writes can also be set explicitly by clients.
* Lookup options
  + Return most recent K values.
  + Return all values in timestamp range (or all values)
* Garbage collection:
  + Only retain most recent K values in a cell
  + Keep values until they are older than K seconds.

**APIs: BigQuery for querying the dataset.**

CRUD Operations no multi row transaction. And since the rows are sorted then the range of queries are fast due to the sequential access.

**Architecture in a client library:**

Master salve model.

It uses Chubby (Zookeeper) coordination service. It is used in the read and write path to get the metadata of a specific data in which tablet server. And then the client contacted that tablet server for writing and reading.

So, it stores a mapping from tablet-to-tablet servers, and it stores the data in a similar data structure of a B+ tree because the rows are sorted.

**Master Node:**

Responsible for managing the tablet servers and the overall data like the schemas. Cluster management, assigning loads and checking the tablets server and it does garbage collection, **but it is not responsible for reading and write paths, so it does not get overloaded.**

All metadata updates go to the master.

**Tablet server:**

managing set of tablets **200 MB** and the tablet server is responsible for n tablets so it’s like 200n MB They are stateless not saving the data because it is done using GFS and the replication is done on GFS level so no need to do it again. It can be divided into large tablet.

Each Tablet server uses LSM tree to store the data where is uses MemTable in memory and stores the SSTable in GFS.

**Read/write Operation:**

Client sends request to chubby asking for a tablet location and it returns the tablet server for it. Then the client contacts the tablet server and does the read operation. The read operation time complexity is O(log n) like in LSM tree but instead of the disk on the machine it will use GFS. And the same for the writer.

**How does Tablet recovery work?**

A tablet server reads its metadata from the metadata table (root node in Chubby)

it reconstructs the state in MemTable from a set of SSTable files by applying pending updates from the REDO log (WAL)

**Spanners**

**Motivation:**

BigTable does not support multi row transactions.

Are globally distributed scalable multi-version database using synchronous replication (using Paxos), externally-consistent (Linearizable) distributed transactions.

Multi version property allows that readers and writers do not block each other. because the reader is reading an old value and the writers are writing a new version of a value.

The main challenge is to build a database that is globally shared, and each shard is also shards into multiple machines and all the shards are replicated. So how to make multirow transactions with linearizable consistency across all the sharded data. And For writing operations it is impossible to take a sync snapshots of all the data that is sharded across multiple data centers and sharded again within a data center.

**What property to achieve?**

Linearizable transactions(External consistency), where after each transaction the up-to-date value must be returned. So all transactions have a sequential order in real time scale.

Spanner builds on many standard techniques:

* State machine replication (Paxos) within a shard
* Two-phase locking (2PL) for serialization
* Two-phase commit (2PC) for distributed transactions or cross-shard transactions
* Multi-version database systems
* Snapshot isolation

**It uses 2PL to achieve Serializability of concurrent transactions:** which means that we can find a serial order of the transactions.

And spanners use a timestamp for all transactions which will lead to a linear order. So, if t1 committed before t2 starts then t1 timestamp is less than t2 timestamp then t2 must see the updated value from t1.

**TrueTime API with uncertainty: it will give interval.**

global clock wall time across machines where is all transactions respect the world clock time then we will get linearizable transactions out of the box. And they assign timestamp to transaction.

So, for concurrent transactions, we choose the commit time in the phase after acquiring the locks and before releasing the locks using 2PL which provides Serializability. So, the two transactions must be disjointed at this time and that gives the order of transactions and even for different transactions not concurrent the same thing will apply.