Database Replication Summary

* An application can work on a single data base server but as the load will increase it will eventually fails. Problem with single data base server(master): -
* If the master fails, whole data is lost
* As write increases, the system will get stuck & even read will also hang.   
  Thus, most user request will time out, i.e. there will be performance issue
* There will be high latency for user connecting from different geo region, i.e. if your server is in New York and user is in Mumbai, i.e. it will not scale
* Advantage of Replication
* protecting your business data against catastrophic events, such as power outages or natural disasters, which could take down an entire cluster
* maintain copies of a product inventory database at two or more separate warehouses, close to the applications that need the data. This feature makes it possible to support massive numbers of clients that could not be supported by a single database instance
* Replication topology
* Master-Slave (Single Leader)
* All writes go to master node. Slave node will pull the changes. If master crashes, then system need to perform **failover** procedure
  + One of the followers need to be promoted to be the new leader,
  + Clients need to be reconfigured to send their writes to the new leader
  + The other followers need to start consuming data changes from the new leader.
* Single master multiple slaves will work if your read is higher than write, i.e. your application is not write intensive such that all writes can be handle by single node
* Advantage:
* **No write-conflict** during concurrent writes
* **High Data integrity**
* Issue:
* Must ensure that single node should be able to handle all the writes
* **Latency could be high** as master database could be in US while client is in India. The write request will have to travel great distance
* **Single point of failure** – If leader crash, then how to elect new leader. Need to define a fail-over strategy. How to select new leader from the set of followers? How the other follower agrees to new leader, i.e. **Consensus Problem**
* **Low availability** as in case of master failure it may take time to elect new master. Now, somehow, we need to ensure that all the new requests from client should go to the new leader, i.e. **request routing**
* Can cause **split-brain** issue when old master was not responsive for a while and it comes back and think itself as the leader. Always choose odd numbers of nodes to avoid split-brain situation, as one part will always have more nodes than other in odd number of nodes cluster
* Multi-Master
* Write can be performed on any master. Although we can have sticky session where a user will be connected to its nearest master database and all write are performed to that database.
* This topology is useful when write and read traffic are almost same and overall writes traffic cannot be handle by single node
* Advantage:
* **write scale** as the write request is evenly distributed across all the master nodes
* **Latency is low** as write can be directed to the master node which is closer to the user
* **high availability –** even a master crash, the other masters continue to take the incoming request
* Issue
  + Should handle **concurrent write** (i.e. write conflict). Conflict happens only during Asynchronous replication
    - There are many ways to resolve conflict one is to last write win, in which each write will have timestamp. During replication when a master node receives an update, if the timestamp of update is higher the timestamp of previous value then applies the changes else discard. Physical clock might node be reliable and when using timestamps, you will probably need at least some sort to clock synchronization, like NTP
    - Another solution is to record these conflicts, and then write application code to allow the user to manually resolve them later
    - Some databases and replication tools allow us to write custom conflict resolution code. This code can be executed on write or on read time. For instance, when a conflict is detected a stored procedure can be called with the conflicting values and it decides what to do with them
  + **DDL replication** 🡪 To perform DDL (adding/removing columns etc.) usually requires a global database lock, wait until all the pending replication take place, and then execute the DDL. In the meantime, all the writes will either be blocked or fail
  + **Causality Problem** 
    - If a master node A inserts a row, and then another master node B updates this row (after taking the replication from node A), but master node C receives the update before the insert, we will have problems. We need to make sure that all the nodes first process the insert event before processing the update event
    - Different ways to solve this problem (for instance, using logical clocks), but the point is: You need to make sure your database or replication tool is handling this issue
  + **Loosely consistent** a client can write to master M1 and read from other master M2, which might not have replicated the latest changes from M2
* There cannot be more than one leader in a data center as it can lead to corruption of data. A single datacenter performs as a single leader follower architecture. A write destined to data center is processed by the leader of that data center
* Leaderless
* The request goes to all the nodes for read/write. The coordinator decided which value to commit or return based on quorum
* In Leaderless architecture **failover** does not exists
* In case of Group Replication, the read goes to one node. Write goes to one node which internally communicate with other nodes in the group and decide whether to commit the changes based on Certification test
* No node is master of any data or table. Write/Read succeed based on quorum-based protocol. Write is send to all the replica & it is considered as success when ack is received from N number of replica
* All the nodes in the cluster keep data in sync using any of these approaches
  + using Read repair
  + Anti-Entropy protocol - each node will pull the changes periodically. Using Merkle tree changes are shared between nodes to find the diff
  + Hinted-hand off
* It should be use when
* system can tolerate occasional stale reads (Eventual Consistency),
* require ***high availability*** and ***low latency***
* Advantage: **No master failure issue**. Hence, all the issues for selecting new leader will not happen here
* A good rule of thumb is to always have w + r > number of replicas
  + For read intensive system, keep value of ‘w’ as high as possible and ‘r’ as low as possible. w = number of replicas, r = 1
  + For write intensive system, keep value of ‘w’ as low as possible and ‘r’ as high as possible. w = 1, r = number of replicas
* Issue:
* Sloppy quorum 🡪 r + w < n, the w writes may end up on different nodes than the r read nodes, so there is no longer a guaranteed overlap between the r read nodes and the w write nodes
* Write succeeds in less than w replicas 🡪 If a write succeeds on some replicas, but failed on others (for example, because the disks on some nodes are full), and overall succeeded on fewer than w replicas, it is not rolled back on the replicas where it succeeded. This means that if a write was reported as failed, subsequent reads may or may not return the value from that write
* Since during write we don’t wait for ack from all the nodes, hence it might be possible that one of the nodes has not updated the record. If we read from that node, then we may get stale data. To overcome this issue, we read from N nodes, and take the value with highest version
* **Concurrent Writes: 🡪** If two writes occur concurrently, it is not clear which one happened first. In this case, the only safe

solution is to merge the concurrent writes. If a winner is picked based on timestamp, writes can be lost due to clock skew

* A replica fails and later restored from s replica with stale data: If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below ***w,*** breaking the *quorum* condition
* To fix the node with stale data
* Read repair
* Anti-entropy process using Merkle Tree (for more info. Check “*System Design\Key Concepts of Distributed System\Hashing\Hashing Techniques\Merkle Tree*”)
* Hinted-hand off
* Consistent Hashing 🡪 All the nodes are arranged in the form of ring.
* Let say replication factor is 3. The key of each record will map to a Node N1. This node will have the primary copy. While next 2 nodes in the ring, i.e. N2 & N3 will have a replica copy of record
* If the node to which the key should be written is down, then hinted hand-off is used. The write operation is written to a hint table/file either at the replica node or coordinator node.
* Once the main node is available the data is replayed from the hint log
* There is threshold for how many write for a node will be stored in the hint log. The threshold is also defined in terms of time or duration till which hint will be stored. After the threshold is reached hint will no longer be updated
* Replication techniques: -
* Synchronous
* The update will first take place on first master, then it will share the changes with all other masters
* When the changes are committed on all the masters/slaves then only the first master will commit it changes. Till all the masters have updated it changes, the client will be blocked.
* Advantage
* **High durability** – It provides strong consistency of data. This is achieved using 2-Phase Commit or 3-Phase Commit protocols
* Issue
* This has **availability and performance** issue. If one replica goes down during update, then write may get blocked
* Semi-Synchronous
* In this case the first master will wait until at least one more master (or X number of masters as configured) has committed the new change.
* Good performance, but if master crash after update or network failure before update could reach to slave then there is risk of data loss
* **Better data durability** than asynchronous mode but **lower performance.** Lower performance is trade-off against data integrity
* The amount of slowdown is at least the TCP/IP roundtrip time to send the commit to the slave and wait for the acknowledgment of receipt by the slave. This means that semi synchronous replication works best for close servers communicating over fast networks, and worst for distant servers communicating over slow networks
* Asynchronous
* The master will update the changes and write it in its binary log and send ack to client without waiting for other masters/slaves to replicates the changes.
* The replicas will read from master binary log as per their convenience
* Advantage:
* High performance
* Issue:
* **Low data durability –** If the node where write occurred crash before the changes are copied to any other node
* **Replication Lag -** If client read from slave database then it may read stale data if update value has been not replicated to the slave
* Can suffer from **write conflict** in master-master replication
* How replication works in Master-Slave & Multi-Master: -
* Master will execute the transaction & update it in binary log file
* Row based 🡪 apply the same changes done on the master to the slave
* If the update has affected 1000 rows, then all the information must be logged and shipped to the slave
* Advantage
  + All changes can be replicated. This is the safest form of replication
  + Fewer row locks are required on the master
  + Fewer row locks are required on the slave for any INSERT, UPDATE, or DELETE statement.
* Disadvantage
  + can generate more data that must be logged. To replicate a DML statement (such as an UPDATE or DELETE statement), statement-based replication writes only the statement to the binary log. By contrast, row-based replication writes each changed row to the binary log
* Statement based
* Only the actual SQL query is transferred to the slave
* Advantage
  + Less data written to log files & transferred over the network
  + Log files contain all statements that made any changes, so they can be used to audit the database
* Disadvantage
  + Not all statements which modify data (such as INSERT

DELETE, UPDATE, and REPLACE statements) can be replicated using statement-based replication

* + Use this, if the query is deterministic, e.g. insert timestamp will give different result every time its executed. Hence, it’s not deterministic
  + Statements using any of the following functions cannot be replicated properly using statement-based replication: RAND(), SLEEP(), VERSION(), LIMIT, ORDER BY etc.
  + If a statement that doubles the amount for a user (initial amount = 1000) is executed twice (for any reasons instead of once) at the masters, then we will have an amount of 4,000 instead of 2,000. Using row-base replication can protect you against this kind of disaster
* Log shipping Replication 🡪 The idea of log shipping replication is to transfer the transaction log files (WAL) to the replicas, that can then apply them to get the exact same result
* It describes the changes at a very low level
* We probably won’t be able to replicate a log generated by a different versions of the database, for example, if the way the data is physically stored may have changed across the versions
* 2 ways of replicating, i.e. copy events from master to slave
* Co-ordinate based replication
* GTID based replication 🡪 each transaction is given unique ID across all the replication servers
* On the slave, there will be I/O & SQL thread
* Slave will connect to Master using I/O thread. For each connection, Master, will spawn a new thread for each slave. I/O thread will then copy the changes from **binary log** of master to its own relay-log
* SQL-thread on the slave will read relay-log and perform the necessary changes to its own local copy
* Slave will read from binary log and update its local copy. Assume during the copying phase the master crashes. Now slave need to copy the changes from another Master. But the binary log of this master will not be in same order as that of previous. To overcome this issue, for each transaction a Global Transaction ID (GTID) is maintained. This enable slave to connect to any master and get the same set of updates. GTID act as a baseline like Clear-case

BUT WHO WILL ASSIGN AND ENSURE EACH GTID IS UNIQUE & MONOTICALLY INCREASING ?

* There are already different products available which solve the above issues, e.g. Galera Cluster Replication, Tungsten Replicator etc. Galera uses more complex algorithm (based on Certification test) to prevent multi-master write conflict. These products work as plugin to MySQL or any other database. They also provide heterogenous replication (i.e. Master is Oracle, Slave could be MySQL)
* Conflict resolution/prevention techniques for Multi-Master: -
* Replication between 2 masters 🡪 Conflict occurs only during asynchronous replication. In synchronous replication changes are done using 2PC or 3PC. Each one will maintain previous and current updated time stamp of each record. These details are shared between masters along with the data change. Using time stamp it can detect if there is write conflict. This logic work if primary key is associated with each record. Refer to conflict resolution in Volt-DB  
  <https://docs.voltdb.com/UsingVoltDB/DbRepHowToActive.php#DbRepConflictProcess>
* Multi-master conflict can also be prevented by ensuring write on a table/schema is happening, only at one place.
* Let say, there are 3 tables T1, T2, T3 and 3 masters M1, M2, M3
* M1 will allow write on T1, while read from T2 & T3
* M2 will allow write on T2, while read from T1 & T3
* M3 will allow write on T3, while read from T1 & T2
* Based on the above configuration, if write for T1 goes to M2 or M3, they will re-direct that request to M1
* This way write will happen at only one place while read can happen from any masters. Refer to Multi-Master Conflict Prevention doc
* e.g. write request for USA customer will go to Datacenter in USA, for Europe customer will go to Europe datacenter. If Europe datacenter receive write from USA customer, it will re-direct the write request to USA datacenter
* GALERA Conflict Resolution
* Each write at the node will be collected in a write set
* The group communication will broadcast the write set to all the nodes
* GALERA can do conflict detection between different write sets, so enqueued (but not yet committed) write sets are protected from local conflicting commits until our replicated write set is committed
* When the write set is applied on a given node, any locking conflicts it detects with open (not-yet-committed) transactions on that node causes that open transaction to get rolled back
* Write sets being applied by replication threads always win
* The implementation of certification-based replication in GALERA Cluster depends on the global ordering of transactions. GALERA Cluster assigns each transaction a global ordinal sequence number, or seq\_no, during replication. When a transaction reaches the commit point, the node checks the sequence number against that of the last successful transaction. The interval between the two is the area of concern, given that transactions that occur within this interval have not seen the effects of each other. All transactions in this interval (are still waiting for being applied at the current node, they are still not committed) are checked for primary key conflicts with the transaction in question. The certification tests fail if it detects a conflict
* The enqueue transaction (which has passed certification test) will be committed at each node asynchronously
* Manually resolve 🡪 store both the value and let user decide which one should be kept and which one to be rejected. This is same as Merge tool we use in Perforce or any version control system
* Trigger store procedure 🡪 there are tools like Bucardo & BDR, which allow code to be written that will be trigger when write conflict is detected. Both the values will be passed to the stored procedure, user can write the logic to resolve it
* ensuring that all transactions for a single user session, or associated with a purchase order, are directed to the same cluster
* Propagation of changes among all nodes to have consistent data across the replicas in Leader less topology
* Anti-entropy (Gossip Protocol) 🡪 some datastores have a **background process** that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. This process is called ***Anti-entropy process***

Each server shares details about it state with neighbor server to keep their data in sync. This can be achieve using Merkle Tree which is hash of all the data record. Each server shares their Merkle tree. Simply comparing the root node, it can detect whether they are in same state or not

* Read repair 🡪 works in quorum-based protocol. When a read request is received for a key = K, data is read from R number of nodes (based on quorum configuration). The latest value is return to the client, while the nodes with older version will update their record with the latest value
* Replication Lag 🡪 The time it takes for updates to be available in all the replicas in case of asynchronous replication. Higher replication lag means user will be unable to see his/her latest changes if the read request goes to non-master node. Different consistency model to overcome this issue
* Read your write 🡪 client will always read the write which he/she has just performed
* Ensure client **read request goes to the leader node** where his last write was performed. For example, if we are implementing something like Twitter, we can read another people’s timeline from a replica (as the user will not be able to write/change it), but when viewing their own timeline, read from the leader, to ensure we don’t miss any update
* If too many writes are happening then all the read request goes to the leader, hence no use of replica
* **Generate a commit token** when you write something to the leader. The client can then send this token to the replica it’s trying to read from, and with this token the replica knows if it’s current enough (i.e. if it has already applied that commit). If so, it can just serve that read without any problem, otherwise it can either block until it receives that update, and then answer the request, or it can just reject it, and the client can try another replica
* Monotonic read 🡪 It guarantees that if you make several reads to a given value, all the successive reads will be at least as recent as the previous one
* monotonic reads can be achieved by ensuring that all the client read requests is send to the same replica
* Every time that a client reads from a replica it receives its latest commit token, that is then sent in the next read, that can go to another replica. This replica can then check this commit token to know if it’s eligible to answer that query (i.e. if its own commit token is “greater” than the one received). If that’s not the case, it can wait until more data is replicated before responding, or it can return an error.
* Bounded staleness consistency 🡪 there should be a limit on how stale the data we are reading is. e.g. we may want to guarantee that clients will not read data that is more than 3 minutes old
* MySQL Group Replication
* Provide in-build plugin which take care of all the server in the group. It is used for Master-Master
* It has shared nothing architecture
* Read can be answered by any server
* For write certification test is performed, all the group servers will be involved and each of they must certified the write transaction after that only write succeed
* No need to manually handle fail over
* GALERA Cluster
* It is based on data centric model, i.e. dataset is synchronized between one or more servers
  + Each data is given a global id which is unique among all the data present in various servers of the clusters
* It works on top of under lying database, i.e. Oracle, MysQL etc.
* All nodes in the clusters are master. Any node can take write/read request
* Each transaction is given a unique Global transaction ID.   
  <DataSet / Cluster ID>:<Seq Number>
* When a write request comes, a write set is prepared (record/primary key whose entry is being changed) and send to all the nodes in the cluster. This is known as certification test. If all the nodes agree, then certification test is passed. Once the certification test has passed, the originator node COMMIT and return to client. All the other nodes execute the pending transaction in the queue and will execute this transaction when it turns arrive
* When a node joins a cluster, a donor is selected which transfer the State of the group using GTID
* If one node in the cluster is slows, then it will take time to certify. Due to it slowness, the pending transaction queue of that node will fill up and this may impact performance of the entire group
* asd