Reading "Designing Data-Intensive Applications"

Replication

Use cases

Use case 1: Increase availability

- Replica could function as hot standby, which could take over immediately if the original component fails.

Use case 2: Increase read throughput

- Increase the number of machine which could serve read queries

Use case 3: Reduce access latency

- Keep data geographically close to your users

When not to use - Scale writes

- No matter what topology you use, all of your writes need to go through a single machine.
    - Although a dual master architecture appears to double the capacity for handling writes (because there are two masters), it actually doesn't. Writes are just as expensive as before because each statement has to be executed twice: once when it is received from the client and once when it is received from the other master. All the writes done by the A clients, as well as B clients, are replicated and get executed twice, which leaves you in no better position than before.

Replication mode

Synchronous

- Def: The master and slaves are always in sync and a transaction is not allowed to be committed on the master unless the slaves agrees to commit it as well (i.e. synchronous replication makes the master wait for all the slaves to keep up with the writes.)
- Pros: Consistency. The follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we could be sure that the data is still on the follower.
- Cons: Performance. If the synchronous follower does not respond, then the leader must block all other writes and white until the synchronous follower is available again.

Asynchronous

- Def: The master does not wait for the slaves to apply the changes, but instead just dispatches each change request to the slaves and assume they will catch up eventually and replicate all the changes.
- Pros: Performance. the transaction is reported as committed immediately, without waiting for any acknowledgement from the slave.
- Cons: Consistency. The follower and the leader will not be in sync.

Semi-synchronous

- Only a subset of followers are configured to be synchrnous and the others are asynchronous. This setting is commonly seen in cross data center replications. The replicas within a single data center are synchronous and the replicas in other data center are asynchronous.

Problems with replication lag

Read your own writes

- Scenario: A user makes a write, followed by a read from replica.
- Read-after-write consistency to rescue. Several possible implementation scenarios:

    i. When reading something that the user may have modified, read it from the leader; Otherwise, read it from a follower. This means you have some way of knowing whether something might have been modified, without actually querying it. For example, user profile informaiton on a social network website is normally only editable by the owner of the profile, not by anybody else. Thus, a simple rule is: Always read the user's own profile from the leader, and any other users' profile from a follower.

    ii. If most things in the application are potentially editable by the user, that approach won't be effective, as most of the things would have to be read from the leader. In that case, other criteria could be used.
        - For example, you could track the time of the last update and, for one minute after the last update, make all reads from the leader. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.
        - The client could remember the timestamp of its most recent write - then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. If a replica is not sufficiently up to date, either teh read can be handled by another replica or the query could wait until the replica has caught up.

- Scenario: The same user is accessing your service from multiple devices, for example a desktop web browser and a mobile app. The user enters information on one device and then views it on another device.

- Cross-device read-after-write consistency to rescue. Several adaptation from read-after-write consistency:
    1. Approaches that require remembering the timestamp of the user's last update become more difficult, because the code running on one device doesn't know what updates have happended on the other device. This metadata will need to be centralized.
    2. If your replicas are distributed across different data centers, there is no guarantee that connections from different devices will be routed to the same datacenter. (For example, if the user's desktop computer uses the home broadband connection and their mobile devices use the cellular network, the devices' network routes may be completely different.) If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same datacenter.

Monotonic reads

- Scenario: A user makes several reads from different replicas and it's possible for a user to see things moving backward in time.
- Monotonic reads consistency to rescue. It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency.
    1. Each user always makes their reads from the same replica. For example, the replica can be chosen based on a hash of the user ID, rather than randomly.

Consistent prefix reads

- Scenario: Mr Poon is asking a question and Mrs Cake is answering it. However, from the observer perspective it could be Mrs Cake is answering the question even before Mr Poon asks for it. The reason is that the things said by Mrs Cake go through a follower with little lag but the things said by Mr. Poons have a longer replication delay.
- Consistent prefix reads to rescue: If a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.
    1. The reason for inconsistency is that different partitions operate independently, so there is no global ordering of writes: when a user reads from the database, they may see some parts of the database in

an order state and some in a newer state. One solution is to make sure that any writes that are casually related to each other are written to the same partition.

Replication Topology

Single leader replication

Responsibility

- Master is reponsible for all data-modifying commands like updates, inserts, deletes or create table statements. The master server records all of these statements in a log file called a binlog, together with a timestamp, and a sequence number to each statement. Once a statement is written to a binlog, it can then be sent to slave servers.
- Slave is responsible for all read statements.

Replication process

- The master server writes commands to its own binlog, regardless if any slave servers are connected or not. The slave server knows where it left off and makes sure to get the right updates. This asynchronous process decouples the master from its slaves - you can always connect a new slave or disconnect slaves at any point in time without affecting the master.
  i. First the client connects to the master server and executes a data modification statement. The statement is executed and written to a binlog file. At this stage the master server returns a response to the client and continues processing other transactions.
  ii. At any point in time the slave server can connect to the master server and ask for an incremental update of the master' binlog file. In its request, the slave server provides the sequence number of the last command that it saw.
  iii. Since all of the commands stored in the binlog file are sorted by sequence number, the master server can quickly locate the right place and begin streaming the binlog file back to the slave server.
  iv. The slave server then writes all of these statements to its own copy of the master's binlog file, called a relay log.
  v. Once a statement is written to the relay log, it is executed on the slave

data set, and the offset of the most recently seen command is increased.

Pros

1. Increase read throughput. Can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves.
2. Increase read availability. Should the master fail, the slaves can still handle read requests.
3. Reduce read latency.

Cons

1. Not help write throughput / availability.
2. Inconsistency between replicas.

Number of slaves

- It is a common practice to have two or more slaves for each master server. Having more than one slave machine have the following benefits:
  - Distribute read-only statements among more servers, thus sharding the load among more servers
  - Use different slaves for different types of queries. E.g. Use one slave for regular application queries and another slave for slow, long-running reports.
  - Losing a slave is a nonevent, as slaves do not have any information that would not be available via the master or other slaves.

Multi-leader replication

Use cases

- Multi-datacenter operations
  - A leader in each data center. Within each datacenter, regular leader-follower replication is used; between datacenters, each datacenter's leader replicates its changes to the leaders in other datacenters.
  - Cons: Need to resolve write conflicts
  - Pros:

	performance	tolerance of datacenter outages	tolerance of network partitions
single-leader config	Every write must go over the internet to the datacenter with the leader. This could add significant latency to writes and might contravene the purpose of having multiple datacenters in the first place	A follower in another datacenter needs to be promoted as leader	single leader replication is sensitive to problems in inter-datacenter link which goes through public network
multi-leader config	Every write can be processed in the local datacenter and is replicated asynchronously to the other datacenters. Thus the internet delay is hidden from users.	Each datacenter could continue operating independently of the others	Could tolerate network interruption better because data is replicated asynchronously

- Clients with offline operation: If you have app that needs to continue working while it is disconnected from the internet.
- Collaborative editing: When one user edits a document, the changes are instantly applied to their local replica and asynchronously replicated to the server and any other users who are editing the same document.

Topology

- Circular / star
    - Pros:
        - Less network traffic.
    - Cons:
        - A read needs to pass through several nodes before it reaches all replicas. If just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed.

- All-to-all topology
    - Pros:
        - Resilient to single node failure
    - Cons:
        - Some network links may be faster than others, could lead to problems discussed in "consistent prefix read"

Leaderless replication

Read repair and anti-entropy

- Scenario: If the client starts reading from the node which failed before and just came back online, then it will get stale data.
- How does it catch up on the writes that it missed:
    1. Read repair: When a client makes a read from several nodes in parallel, it can detect any stale response. If the client sees that a replica has stale value, the client could write newer value back to that replica.
    2. Anti-entropy process: Some datastores have a backend anti-entropy process constantly looking for differences in the data between replicas and copies any missing data from one replica to another.

Quorums for reading and writing

- Rule: If there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each node. As long as w + r > n, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date. Reads and writes that obey these r and w values are called quorum reads and writes. Normally, reads and writes are always sent to all n replicas in parallel. The parameters w and r determine how many nodes we wait for - how many of n nodes need to report success before we consider the read or write to be successful.
- Reasoning: The set of nodes to which you've written and the set of nodes from which you've read must overlap. That is, among the nodes you read there must be at least one node with the latest value.
- Limitation: Even with r + w > n, there are likely to be edge cases where stale values are returned.

i. If sloppy quorum is used, the w writes may end up on different nodes than the r reads, so there is no longer a guaranteed overlap between the r nodes and the w nodes.
ii. If two writes occur concurrently and the winner is picked based on a timestamp, writes could be lost due to clock skew.
iii. If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it is undetermined whether the read returns the old or new value.
iv. If a write succeeded on some replicas but failed on others, and overall succeeded on fewer than w replicas, it is not rolled back on the replicas where it succeeded.
v. 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 might fall below w, breaking the quorum condition.

Sloppy quorums and hinted handoff

- Def

    - Sloppy quorum: Writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n "home" nodes for a value. This typically happens in a large cluster with significantly more than n nodes. It is likely that the client can connect to some database nodes during the network interruption, just not to the nodes that it needs to assemble a quorum for a particular value. If it still decides to accept writes anyway, and write them to some nodes that are reacheable but are not among the n nodes on which the value actually exists.
    - Hinted handoff: Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes.

- Use case: Increasing write availability: as long as any w nodes are available, the database can accept writes. It isn't a quorum at all in the traditional sense. It's only an assurance of durability.

Detecting concurrent writes

- Conflict avoidance

    - Def: If the application ensure that all writes for a particular record go through the same leader, then conflicts cannot occur.
    - Example case: In an application where the user could edit their own data, you can ensure that requests from a particular user are always routed to the same DC and use the leader in that DC for reading and writing; However, in cases when one DC fails or the user changes the location, this approach could break.
- Last write wins. LWW achieves the goal of eventual convergence, but at the cost of durability. There are several popular approaches:
    - Gives each write a unique id (timestamp, random number, a hash of the key and value) and then pick the write with the highest ID as the winner and throw away the other writes. This approach suffers from data loss.
    - Give each replica a unique ID, and let writes that originated at a higher numbered replica always take precedence over writes that originated at a lower numbered replica. This approach suffers from data loss.
- Merging concurrently written values
    - An example for adding things: Shopping cart.
    - An example for deleting things: Tombstone.
- Version vector
    - A version number is kept per replica as well as per key. Each replica increments its own version number when processing a write,, and also keeps track of the version numbers it has seen from each of the other replicas. This information indicates which values to override and which values to keep as siblings.
- Custom conflict resolution logic
    - Record the conflict in an explicit data structure that preserves all information, and write application code that resolves the conflict at some later time.
- Question: [TODO] How does Amazon resolves the conflict within the shopping cart?

Partitioning

Use cases

- Scale horizontally to any size. Without partitioning, sooner or later, your data set size will be too large for a single server to manage or you will get too many concurrent connections for a single server to handle. You are also likely to reach your I/O throughput capacity as you keep reading and writing more data. By using application-level sharing, none of the servers need to have all of the data.

Partitioning by database primary key

Range partitioning

- Def: Assign a continuous range of keys (from some minimum to some maximum) to each partition.
- Pros:
  - Range queries are easy because keys are kept in sorted order within each partition.
- Cons:
  - Certain access patterns could lead to hotspots. e.g. In IoT world, if the key is a timestamp and partitions correspond to ranges of time (one partition per day), then all the writes end up going to the same partition. To solve the problem, you could prefix each timestamp with the sensor name so that the partitioning is first by sensor name and then by time.

Hash partitioning

- Def: Computes a hash of the input in some manner (MD5 or SHA-1) and then uses modulo arithmetic to get a number between 1 and the number of the shards.
- Pros:
  - It could help reduce hot spots problems.
    - Note: It cannot avoid hot spots problems completely. For example, on a social media site, a celebrity user with millions of followers may cause a storm of activity when they do something. Today most database systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew. For example, a simple technique is to add a random number to the beginning or

end of the key.

- Cons:
  - Range queries are not efficient because once adjacent keys are now scattered across all the partitions.

Consistent hashing

- Def: The entire hash range is shown as a ring. On the hash ring, the shards are assigned to points on the ring using the hash function. In a similar manner, the rows are distributed over the ring using the same hash function. Each shard is now responsible for the region of the ring that starts at the shard's point on the ring and continues to the next shard point. Because a region may start at the end of the hash range and wrap around to the beginning of the hash range, a ring is used here instead of a flat line. The most commonly used functions are MD5, SHA and Murmur hash (murmur3 -2^128, 2^128)
- Pros:
  - Less data migration: gauranteed to move rows from just one old shard to the new shard.
- Cons:
  - Easy to distribute unevenly and hard to balance node
    - Unbalanced scenario 1: Machines with different processing power/speed.
    - Unbalanced scenario 2: Ring is not evenly partitioned.
    - Unbalanced scenario 3: Same range length has different amount of data.
  - Virtual nodes (Used in Dynamo and Cassandra)
    - Solution: Each physical node associated with a different number of virtual nodes.
    - Problems: Data should not be replicated in different virtual nodes but the same physical nodes.

Concatenated index

- Def: Cassandra achieves a compromise between the range and hash partitioning strategy. A table in Cassandra can be declared with a compound primary key consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are

used as a concatenated index for sorting the data in Cassandras SSTables.
- Pros: A query therefore cannot search for a range of values within the first column of a compound key, but if it specifies a fixed value for the first column, it could perform an efficient range scan over the other columns of the key.

Partitioning by database secondary indexes

- Def: A secondary index usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value: find all actions by user 123, find all articles containing the word hogwash.

Partitioning secondary indexes by document

- Def: Each partition maintains its own secondary indexes, covering only the documents in that partition. (Local index)
  - Whenever you write to the database - to add, remove, or update a document - you only need to deal with the partition contains the document ID that you are writing.
  - Reading from a document partitioned index requires span across several different partitions. This approach to querying a partitioned database is known as scatter/gather, which can make read queries quite expensive.

Partitioning secondary indexes by term

- Def: A global index which covers data in all partitions.
  - Reads are more efficient. Rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants.
  - Writes are slower and more complicated because a write to a single document may now affect multiple partitions of the index. Furthermore, in the ideal world, the index would always be up to date, and every document written to the database would immediately be reflected in the index. However, in a term-partitioned index, that would require a distributed transaction across all partitions affected by a write.

Rebalancing strategies

Fixed number of partitions

- Def: Create many more partitions than there are nodes and assign several partitions to each node. When a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again.
- Pros/Cons: This approach is widely used in Riak, ElasticSearch, Couchbase and Voldemort. Choosing the right number of partitions is difficult if the total size of the dataset is highly variable. Since each partition contains a fixed fraction of the total data, the size of each partition grows proportionally to the total amount of the data in the cluster.
    - If partitions are very large, rebalancing and recovery from node failures become expensive.
    - If partitions are too small, they incur too much overhead.

Dynamic partitioning

- Def: When a partition grows to exceed a configured size, it is split into two partitions so that approximately half of the data ends up on each side of the split. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it could merge with an adjacent partition.
- Tradeoffs
    - Pros: The number of partitions adapts to the total data volume.
    - Cons: There is usually no priori info about where to draw the partition boundaries. All writes have to be processed by a single node while the other sit idle.

Partitioning proportionally to nodes

- Def: Make the number of partitions proportional to the number of nodes - to have a fixed number of partitions per node. The size of each partition grows proportionally to the dataset size while the number of nodes remained unchanged. When you increase the number of nodes, the partitions become small again.
- Used by Cassandra and Ketama

Request routing

Common approaches

1. Allow clients to contact any node
2. Send all requests from client to a routing tier first
3. Require that clients be aware of the partitioning and the assignmnet of partitions to nodes.

How does dynamic routing changes get reflected

- Many distributed systems rely on a separate coordination service such as ZooKeeper to keep track of this cluster metadata. Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of partitions to nodes. Other actors, such as the routing tier or the partitioning-aware client, can subscribe to this information within ZooKeeper. Wheneer a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.

Challenges

Cross-shard joins

- Tricky to execute queries spanning multiple shards. The most common reason for using cross-shard joins is to create reports. This usually requires collecting information from the entire database. There are basically two approaches to solve this problem
  - Execute the query in a map-reduce fashion (i.e., send the query to all shards and collect the result into a single result set). It is pretty common that running the same query on each of your servers and picking the highest of the values will not guarantee a correct result.
  - Replicate all the shards to a separate reporting server and run the query there. This approach is easier. It is usually feasible, as well, because most reporting is done at specific times, is long-running, and does not depend on the current state of the database.

Using AUTO_INCREMENT

- It is quite common to use AUTO_INCREMENT to create a unique identifier for a column. However, this fails in a sharded environment because the the shards do not syncrhonize their AUTO_INCREMENT identifiers. This means if you insert a row in one shard, it might well happen that the same identifier is used on another shard. If you truly want to generate a unique identifer, there are basically three approaches.
    - Generate a unique UUID. The drawback is that the identifier takes 128 bits (16 bytes).
    - Use a composite identifier. Where the first part is the shard identifier and the second part is a locally generated identifier. Note that the shard identifier is used when generating the key, so if a row with this identifier is moved, the original shard identifier has to move with it. You can solve this by maintaining, in addition to the column with the AUTO_INCREMENT, an extra column containing the shard identifier for the shard where the row was created.
    - Use atomic counters provided by some data stores. For example, if you already use Redis, you could create a counter for each unique identifier. You would then use Redis' INCR command to increase the value of a selected counter and return it with a different value.

Distributed transactions

- Lose the ACID properties of your database as a whole. Maintaining ACID properties across shards requires you to use distributed transactions, which are complex and expensive to execute (most open-source database engines like MySQL do not even support distributed transactions).

Consistency

Linearizability (Total order)

- Def: To make a system appear as if there were only one copy of the data and all operations on it are atomic.
- Examples:
    - Single leader replication is potentially linearizable if all reads are made from the leader, or from synchronously updated followers, they have the potential to be linearizable.

    - Consensus algorithms such as ZooKeeper and etcd are linearizable.
    - Multi-leader replication are generally not linearizable because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes.
    - A leaderless system with Dynamo-style replication is generally not linearizable.
    - "Last write wins" conflict resolution methods based on time-of-day clocks are almost certainly nonlinearizable because clock timestamps cannot be guaranteed to be consistent with actual event ordering due to clock skew.

CAP theorem

- History: Applications don't require linearizability can be more tolerant of network problems (CAP)
- Def: Either consistent or availability when partitioned.
    - Consistency: Every read would get the most recent write.
    - Availability: Every request received by the nonfailing node in the system must result in a response.
    - Partition tolerance: The cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other.
- Limitations:
    - It only considers one consistency model (namely linearizability) and one kind of fault (network partitions). It doesn't say anything about network delays, dead nodes, or other trade-offs. Influential theoretically, but does not provide any practical guide.

Casual consistency

- Causality provides us with a weaker consistency model when compared with libearizability. Some things can be concurrent, so the version history is like a timeline with branching and merging. Causal consistency does not have the coordination overhead of linearizability and is much less sensitive to network problems.
- Implementation:
    - Linearizability. Any system that is linearizable will preserve causality

    correctly. However, it comes at performance cost.
  - Version vectors. However, keeping track of all causality can be impractical.
  - Lamport stamp: A method of generating total ordering sequence ordering numbers
    - Def: Simply a pair of (counter, node ID).
    - Why consistent with causality: Every onde and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.
    - Limitations: The total order of operations only emerges after you have collected all of the operations. It's not sufficient to have a total ordering of operations - you also need to know when that order is finalized. Total order broadcast to rescue

Total order broadcast

- Def: Total order broadcast implies two properties
  - Reliable delivery: No messages are lost: if a message is delivered to one node, it is delivered to all nodes.
  - Totally ordered delivery: Messages are delivered to every node in the same order.
- Usage:
  - To implement serializable transactions
  - To create a replication log
  - To implement a lock service that provides fencing tokens

Consensus algorithm

Consensus equivalent problems

- Linearizable compare-and-set registers
- Atomic transaction commit
- Total order broadcast
- Locks and leases

- Membership/Coordination services
- Uniqueness constraint

Categories

Non-fault tolerant consensus: Two phase commit and XA transactions

- Def: Two phase commit is an algorithm for achieving atomic transaction commit across multiple nodes - To ensure that either all nodes commit or all nodes abort.
- The process: The protocol contains two crucial points of no return: When a participant votes yes, it promises that it will definitely be able to commit later; And once the coordinator decides.
- Why not-fault tolerant: If the coordinator fails,
  - 2PC must wait for the coordinator to recover, and accept that the system will be blocked in the meantime.
  - Manually fail over by getting humans to choose a new leader node and reconfigure the system to use it.

Fault-tolerant consensus : VSR, Paxos, Raft and Zab

- Def: A fault tolerant consensus satisfy the following properties:
  - Uniform agreement: No two nodes decide differently.
  - Integrity: No node decides twice.
  - Validity: If a node decides value v, then v was proposed by some node.
  - Termination: Every node that does not crash eventually decides on some value.
- Assumptions:
  - The termination property is subject to the assumption that fewer than half of the nodes are crashed or unreacheable.
  - There are no Byzantine faults.
  - Requires at least a majority of nodes to be functioning correctly in order to assume termination.
- Costs:
  - Within the algorithm, the process by which nodes vote on proposals before they are decided is a kind of synchronous replication. However, in practice database are often configured to use asynchronous

    replication.
    - Consensus systems always require a strict majority to operate.
    - Consensus algorithms generally rely on timeouts to detect failed nodes. In environments with highly variable network delays, it could result in frequent leader elections and terrible performance.
    - Sometimes consensus algorithms are sensitive to network problems.
- Could consider use a tool like ZooKeeper to provide the "outsourced" consensus, failure detection and membership service.

Update consistency

- Def: Write-write conflicts occur when two clients try to write the same data at the same time. Result is a lost update.
- Solutions:
    - Pessimistic approach: Preventing conflicts from occuring.
        - The most common way: Write locks. In order to change a value you need to acquire a lock, and the system ensures that only once client can get a lock at a time.
    - Optimistic approach: Let conflicts occur, but detects them and take actions to sort them out.
        - The most common way: Conditional update. Any client that does an update tests the value just before updating it to see if it is changed since his last read.
        - Save both updates and record that they are in conflict. This approach usually used in version control systems.
- Problems of the solution: Both pessimistic and optimistic approach rely on a consistent serialization of the updates. Within a single server, this is obvious. But if it is more than one server, such as with peer-to-peer replication, then two nodes might apply the update in a different order.
- Often, when people first encounter these issues, their reaction is to prefer pessimistic concurrency because they are determined to avoid conflicts. Concurrent programming involves a fundamental tradeoff between safety (avoiding errors such as update conflicts) and liveness (responding quickly to clients). Pessimistic approaches often severly degrade the responsiveness of a system to the degree that it becomes unfit for its purpose. This problem is made worse by the danger of errors such as deadlocks.

Read consistency

- Def:
  - Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write.
- Types:
  - Logical consistency: Ensuring that different data items make sense together.
    - Example:
      - Martin begins update by modifying a line item
      - Pramod reads both records
      - Martin completes update by modifying shipping charge
  - Replication consistency: Ensuring that the same data item has the same value when read from different replicas.
    - Example:
      - There is one last hotel room for a desirable event. The reservation system runs onmany nodes.
      - Martin and Cindy are a couple considering this room, but they are discussing this on the phone because Martin is in London and Cindy is in Boston.
      - Meanwhile Pramod, who is in Mumbai, goes and books that last room.
      - That updates the replicated room availability, but the update gets to Boston quicker than it gets to London.
      - When Martin and Cindy fire up their browsers to see if the room is available, Cindy sees it booked and Martin sees it free.
  - Read-your-write consistency (Session consistency): Once you have made an update, you're guaranteed to continue seeing that update. This can be difficult if the read and write happen on different nodes.
    - Solution1: A sticky session. a session that's tied to one node. A sticky session allows you to ensure that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too. The downsides is that sticky sessions reduce the ability of the load balancer to do its job.

        - Solution2: Version stamps and ensure every interaction with the data store includes the latest version stamp seen by a session.

Replication Consistency

- Def: Slaves could return stale data.
- Reason:
    - Replication is usually asynchronous, and any change made on the master needs some time to replicate to its slaves. Depending on the replication lag, the delay between requests, and the speed of each server, you may get the freshest data or you may get stale data.
- Solution:
    - Send critical read requests to the master so that they would always return the most up-to-date data.
    - Cache the data that has been written on the client side so that you would not need to read the data you have just written.
    - Minize the replication lag to reduce the chance of stale data being read from stale slaves.