Assignment 4

**1. Kleppmann Chap 5**

**Explain how version numbers are used in the Shopping card example (buying food) when capturing the happens-before relationship to know when to merge in values from “concurrent writes”?**

The "happens-before" relationship is a way to determine if one operation occurs before another in a sequence. If operation B builds upon operation A or is aware of it, then we say A "happens before" B.

Two operations are considered concurrent if neither operation knows about the other at the time it starts.

The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written. When a client reads a key, the server returns all values that have not been over‐ written, as well as the latest version number.

When a client writes a key, it includes the version number from the prior read and merges together all values that it received in the prior read.

When the server receives a write with a particular version number, it can overwrite all values with that version number or below since it knows that they have been merged into the new value. But it keeps all values with a higher version number (because those are concurrent with the incoming write).

If a write does not include a version number, then it is concurrent with all other writes and will not overwrite anything and be returned as one of the values on subsequent reads.

Summary of the algorithm is as follows

1. **Version Management**: The server assigns a version number to each key, incrementing it with each write and storing the version alongside the new value.
2. **Read Process**: When a client reads a key, the server returns all non-overwritten values and the latest version number. Clients must read a key before writing to it.
3. **Write Process**: When writing, the client includes the version number from its prior read and merges all values received in that read. The server response can include all current values, enabling chained writes.
4. **Conflict Resolution**: Upon receiving a write, the server overwrites values with the same or lower version numbers, as they’ve been merged, but retains higher version numbers to handle concurrent writes.

**2. Kleppmann Chap 6**

**a) What is the best way of supporting re-partitioning? And why is this the best way? (According to Kleppmann).**

The process of moving load from one node in the cluster to another is called rebalancing.

Fixed number of partitions where there are many more partitions than nodes is the best way to support re-partitioning according to Kleppmann.

If a node is added to a cluster, the new node *steals* a few partitions from other nodes until partitions are fairly distributed again. Only entire partitions are moved between nodes, the number of partitions does not change, nor does the assignment of keys to partitions.

This method is good because of the following reasons:

* When a new node is added, it "steals" some partitions from existing nodes rather than requiring all data to be redistributed.
* When a node is removed, it redistributes its partitions to the other nodes.
* read and write operations can continue uninterrupted during rebalancing, as the new assignment only takes effect once the data transfer is complete.
* A fixed number of partitions remains constant. This approach is operationally simpler than dynamically splitting and merging partitions, as it avoids the complexity and overhead associated with constant partition management.

**b) Explain when you should use local indexing, and when you should use global indexing?**

Local indexes are secondary indexes that are stored in the same partition as the primary key and value. Global indexes, on the other hand, are secondary indexes that are partitioned separately from the primary index, using the indexed value as the partitioning key. This means that a global index may include records from all partitions of the primary key.

Use local index when write throughput more important than read throughput.

Use global index when read throughput is more important than write throughput.

**3. Kleppmann Chap 7**

**a) Read committed vs snapshot isolation.**

**We want to compare read committed with snapshot isolation. We assume the traditional way of implementing read committed, where write locks are held to the end of the transaction, while read locks are set and released when doing the read itself. Show how the following schedule is executed using these two approaches:**

**r1(A); w2(A); w2(B); r1(B); c1; c2;**

Read committed makes sure that dirty reads and dirty writes do not occur.

* Write locks until the transaction commits.
* Read locks are set only when reading and released immediately after the read.

Snapshot isolation makes sure that every transaction reads from a consistent snapshot of the database (the data at the start of the transaction).

Read committed approach.

r1(A); w2(A); w2(B); c2; r1(B); c1;

* Because of dirty reads, r1(B) must wait until c2 has committed.

Snapshot isolation approach:

r1(A); w2(A); w2(B); r1(B); c1; c2;

* Transaction 1 is unaffected by the concurrent writes from transaction 2 because it referrers to the snapshots. But transaction 1 reads the old value of B, and not the updated value written by transaction 2.

**b) Also show how this is executed using serializable with 2PL (two-phase locking).**

Serializable two-phase locking:

r1(A); r1(B); c1; w2(A); w2(B); c2;

* Transaction 1 reads and commits before transaction 2 because transaction 1 has a read lock on A that is only released on commit and transaction 2’s read of A must wait until the transaction 1’s commit.

**4. Kleppmann Chap 8**

**a) If you send a message in a network and you do not get a reply, what could have happened? List some alternatives.**

1) it could have reached the destination and is still being processed (response will come later).

2) it could have reached the destination, but the response did not reach back because of some faults (response won’t be received).

3) The message you sent did not reach the destination and was lost on its way.

4) The response could be on its way back and is delayed because of a congestion in your own network.

**b) Explain why and how using clocks for last write wins could be dangerous.**

This is dangerous because clocks on different machines have drifts (higher or lower than the actual time). This drift can cause the first write to be labeled as last write and vice-versa. Also in the case where two writes have the same timestamp, a tie breaker needs to be used which can lead to violations of causality.

**c) Explain how a node sometimes cannot trust its own judgement**

For example, if a node experiences a stop-the-world garbage collector pause, after it resumes the node continues as if nothing ever happened.

Another example could be that a node can receive incoming messages, but its outgoing messages are dropped or delayed. the node may function correctly from its perspective, but other nodes perceive it as dead due to the lack of communication. The isolated node is then powerless to prevent being incorrectly declared dead by the rest of the system.

**5. Kleppmann Chap 9**

**a) Explain the connection between ordering, linearizability and consensus.**

Ordering : Ordering is a fundamental idea that is essential for preserving causality. Causality here refers to the relationship between cause and effect.

Linearizability: Is a form of recency guarantee where there is only total order (No concurrent events, where two events must happen one after another) of operations. It also acts as if there is only 1 copy of the data where every operation is atomic. Linearizability implies causality, in other words, any system that is linearizable will preserve causality order.

Consensus: Consensus is a fundamental problem in distributed systems that involves getting multiple nodes to agree on a single outcome. It's essential for tasks like leader election, where all nodes must agree on the current leader to avoid data inconsistencies. Nodes must agree on critical states like which node is the leader, which value is committed, or which sequence of operations to execute. For consensus to work effectively, it’s not enough that nodes simply agree on a single outcome. They also need to agree on the order in which decisions are made. The connection with linearization and consensus is that both are atomicity.

b) **Are there any distributed data systems which are usable even if they are not linearizable?**

Distributed data systems that prioritize availability, fault tolerance and performance tend to avoid strict linearizability.

**c) What is the main reason for not having a linearizable distributed system?**

The main reason for not having a linearizable distributed system is the performance and scalability cost associated with achieving linearizability, especially in terms of latency and availability. It often necessitates synchronizing across multiple nodes before completing an operation.

In the presence of network delays, a read operation might need to wait for a potentially slow write operation to complete.

**6. Coulouris Chap 14**

**a) Given two events e and f. Assume that the logical (Lamport) clock values L are such that L (e)< L (f). Can we then deduce that e "happened before" f? Why? What happens if one uses vector clocks instead? Explain.**

L (e)< L (f) suggests that e may have happened before f, but this is not certain. Lamport clocks only ensure that if e → f then L(e)<L(f), but the reverse is not necessarily true. The reason for this is because it relies on a path from one event to another in the causal space to decide if one event occurred before another. If both e and f are in the same process then yes e → f but if not, its not guaranteed.

A diagram of a line with points and letters

Description automatically generated with medium confidence

For example, in the figure above, there is no causal relation between b and e, b and e are concurrent. We see that L(e) < L(b) but e||b (they are concurrent).

A diagram of a graph

Description automatically generated

Unlike the Lamport clock system the vector clock has the useful property of if L(e)<L(f) then e → f. In the example of concurrency, we used before between b and e, we can see that this is false using vector timestamp comparison rules, this means that b and e are concurrent.

**b) The figure below shows three processes and several events. Vector clock values are given for some of the events. Give the vector clock values for the remaining events.**

A diagram of a diagram

Description automatically generated

A diagram of a diagram

Description automatically generated

Separate the digits in red with a comma (,) and add parenthesis () to mark them as a vector (hard to draw with mouse).

c) **The figure below shows the events that occur at two processes P1 and P2. The**

**arrows mean sending of messages. Show the alternative consistent states the system**

**can have had. Start from state S00. (Sxy where x is p1's state and y is p2's state)**

A graph of a triangle

Description automatically generated with medium confidence

**7. RAFT**

**RAFT has a concept where the log is replicated to all participants. How does RAFT ensure that the log is equal on all nodes in case of a crash and a new leader?**

* Each Raft cluster has a single leader at any given time. If a leader fails, a new leader is elected.
* Raft ensures that if two entries in different logs have the same index and term, they store the same command, and all preceding entries are identical. This is known as the *Log Matching Property*. (used to detect inconsistencies and synchronize the logs when a new leader is elected)
* Heartbeats/AppendEntries are sent from the leader to all followers periodically to replicate new log entries and confirm committed entries. It includes information about the preciding log entry (index and term) on the leader. Followers receive this and check if their log matches the leader’s (based on the previous index and term). If there is a mismatch, the follower identifies the conflict and the leader helps to delete bad entries and append the correct ones.
* A leader candidate must have a log that has the highest term or the longest log (if terms are equal). When a node becomes the new leader it begins sending AppendEntries to synchronize all followers.
* Only entries that have been acknowledged by a majority of nodes can be committed and applied to the state machine. A new leader ensures all nodes replicate at least up to the last committed entry. By ensuring majority replication before committing, Raft guarantees that a committed log entry remains durable, even if some nodes fail or a new leader is elected.

A more condensed answer for the question above is:

1. *Election of the New Leader*: A node with the most up-to-date log (by term or length) is elected as the new leader.
2. *Log Consistency Check*: The new leader sends AppendEntries to followers to synchronize logs, overwriting inconsistent entries.
3. *Majority Commitment*: The new leader ensures that only entries replicated on a majority are marked as committed.
4. *Followers’ Synchronization*: Any follower with missing or inconsistent entries adjusts its log to match the new leader's log.

**8. MySQL RAFT**

**What are the main takeaways (advantages) from introducing RAFT into MySQL?**

The main advantage is guaranteeing safety and avoiding data loss during the complex promotion and failover operations. This makes control plane operations like promotions and membership changes (adding/removing nodes) safe.

The main takeaways are that node changes (promotions and membership) become much simpler. The time to do takeover was reduced from 20-40 seconds to approx. 2 seconds when using built-in RAFT

**9. Stonebraker/Pavlo**

**What is happening with Document vs SQL databases according to Stonebraker/Pavlo?**

According to Stonebraker and Pavlo, document databases, particularly those associated with the NoSQL movement, have primarily appealed to developers for their JSON support, which aligns with the structure of web-based applications. Proponents argue that document DBMSs simplify development by reducing the "impedance mismatch" between object-oriented code and relational databases, allowing nested data to be stored directly and eliminating the need for multiple SQL joins to fetch related data. This approach is often seen as solving the “N+1 problem” common in ORM frameworks. However, Stonebraker and Pavlo critique this model, noting issues with data redundancy, lack of data independence, and the performance limitations of prejoined or denormalized structures.

Despite initial resistance to SQL, by the end of the 2010s, nearly all NoSQL databases (including document DBMSs) introduced SQL-like query interfaces, sometimes to meet performance needs or developer preferences. As SQL relational databases adopted JSON capabilities and NoSQL databases added SQL-like interfaces, Stonebraker and Pavlo anticipate that distinctions between these systems will narrow further