TDT4225 Very Large, Distributed Data Volumes Exercise 4

Fredrik Sundt-Hansen

November 2024

1 Kleppmann Chap 5

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

The server assigns a version number to each key and increments this number with every write operation to that key. Before writing, a client must first read the key. During the read, the server returns all values that have not been overwritten, along with the latest version number. When a client writes a key, it includes the version number obtained from the prior read and merges all values received during that read. Upon receiving a write request with a specific version number, the server overwrites all values associated with that version number or lower, while retaining any values with higher version numbers.

2 Kleppmann Chap 6

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

According to Kleppmann, the best way of supporting re-partitioning is to use **a fixed number of partitions**. This involves creating many more partitions than there are nodes, and assign several partitions to each node. The advantages of this approach, easier partitioning. One node may steal some partitions from existing nodes til the partitions are fairly distributed, and the same happens if a node is removed from the cluster (just in the opposite way). Additionally, by assigning more partition to nodes that are more powerful, the share load can be more fairly distributed. Lastly, the use of fixed partitions have the advantage of being simpler to set up and manage later, as the number of partitions does not change later.

On the other hand, there are some disadvantages to this approach. There is limited scalability. As mentioned above, the number of partitions are configured ad set at the start, which also means that this is the maximum of nodes one can have. This introduces the challenge of choosing the right number of partitions,

especially if the dataset size if variable. By choosing too many partitions, this introduces too much overhead.

However, Kleppman argues that using a fixed number of partitions is a good balance between simplicity, scalability and performance.

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

Local indexes are a good choice when write speed is more important than read speed, and when queries in the secondary index are rare. On the other hand, **global indexes** are a good choice when the opposite takes place. That is when read speed is more important than write speed, and queries on secondary index are more frequent.

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:

1. **r1(A)** Transaction 1 reads A, a read lock is set on A and is released immediately after reading.
2. **w2(A** Transaction 2 wants to write to A, it has to wait for transaction 1 to be done. Transaction 2 gets a write lock on A.
3. **w2(B** Transaction 2 writes to B, gets a write lock on B.
4. **r1(B** Transaction 1 tries to read B, but has to wait for transaction to release its write lock, to avoid dirty writes.
5. **c1** Transaction 1 finishes and releases write lock on A.
6. **c2** Transaction 2 finises and releases write lock on A and B, Transaction 1 can now read B.

Snapshot isolation:

1. **r1(A)** Transaction 1 starts and takes a snapshot of the database. It then reads A from this snapshot.
2. **w2(A)** Transaction 2 writes to A. This write is not visible for transaction 1, as it reads from a previous snapshot.

3. **w2(B)** Transaction 2 writes to B, this transaction is neither visible for transaction 1.

4. **r1(B)** Transaction 1 reads from its snapshot. It reads older values of B, as the changes from transaction 2 are not visible yet.

5. **c1** Transaction 1 finishes.

6. **c2** Transaction 2 finishes.

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

1. **r1(A)** Transaction 1 reads A and gets lock on A.

2. **w2(A)** Transaction 2 wants to read A, but has to wait due to the transaction 1 has acquired the lock. Transaction 2 is blocked til transaction 1 finishes, it reads and releases the lock.

3. **w2(B)** Transaction 2 writes to B and gets an lock on B.

4. **r1(B)** Transaction 1 wants to read B, but has to wait on transaction 2 to finish and release the lock on B.

5. **c1** Transaction 1 finishes and releases the lock on A.

6. **c2** Transaction 1 finishes and releases the lock on A and B. Now can transaction 1 read.

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.

- **The message could have gotten lost.** For example, the network was disconnected.
- **Message may be waiting in a queue, and is delayed**
- **The remote node may have failed** Perhaps it creases or on outage lead to it was powered down.
- **The remote node may have temporarily stopped,** It may have experienced a long garage collection pause, but it will start again later.
- **The response could have gotten lost.** The node may have processed the message, but the response has been lost on the network.
- **The response is delayed**

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

Lat Write wins (LWW) is a strategy for conflict resolution in distributed databases, where the newest write is the correct version. One can use timestamps to decide which strategy is the newest one, but there are some caveats:

- **Synchronization** Clocks on nodes are rarely perfect. This can lead to the scenario where a write at a later time is wrongly assumed as older than a previous write, which can lead to data loss.
- **LWW cannot distinguish between writes that occurred sequentially un quick succession and writes that were truly concurrent**
- **Duplicated timestamps** It is possible to two nodes to generate writing at the same timestamp. To resolve this, extra logic needs to be implemented.
- **Loss of data** A node with a clock which is behind, cannot overwrite values that has been written earlier by a clock that is too fast. This can lead to big loss of data without the system warning about the error.

c) Explain how a node sometimes can not trust its own judgement.

In distributed systems a node can sometime not trust its own judgement, due to the uncertainty in the communication over the network and potentially for partials fails. There can be multiple scenarios where this might occur:

- **Asymmetric failure** A node can receive messages, but all outgoing messages from a node are either dropped or delayed. Even though the node is fully functional, the other nodes can node receive its responses, and the sending node is wrongly accused as dead after a given time.
- **Pause in garbage collection** A node can receive long pauses doe to garbage collection, where all threads are stopped in a time frame. Other nodes can accuse the missing response as an error, and declare it as dead.
- **Partial disconnected node** A node might end up being disconnected to parts of the network, which results in that it does not receive important messages, for example the choice of a new leader.

5 Kleppmann Chap 9

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

Linearizability implies ordering: To achieve linearizability, the system must guarantee a total ordering of operations. This ordering ensures that all nodes see the same sequence of events, avoiding conflicts or inconsistencies.

Consensus is necessary for linearizability: In practice, achieving linearizability without consensus algorithms is challenging. For example, in a leader-based replication system, nodes must agree on who the leader is to prevent "split-brain" scenarios and ensure a consistent ordering of writes.

Total ordering is equivalent to consensus: Total order broadcast is a mechanism for delivering messages in the same order to all nodes, which is essential for replicating data consistently. It turns out that total order broadcast is equivalent to consensus: solving one problem enables you to transform the solution to solve the other.

b) Are there any distributed data systems which are usable even if they are not linearizable? Explain your answer.

Yes, there are many distributed data systems that are usable even without being linearizable.

- Eventually consistent systems: Many "NoSQL" databases, such as Cassandra and Riak, are designed to be eventually consistent. This means that data across different nodes may be temporarily inconsistent but will eventually converge to a consistent state. These systems often prioritize high availability and performance over strict consistency.
- Multi-leader replication: Systems that use multi-leader replication are typically not linearizable because they handle writes simultaneously on multiple nodes and replicate them asynchronously to other nodes. This can lead to conflicts that require resolution but offers advantages in availability and managing data across multiple data centers.
- Leaderless systems: Systems with leaderless replication, such as Dynamo-inspired databases, generally do not provide linearizability. While they can offer "strong consistency" through quorum reads and writes, non-linearizable behavior is still possible, particularly when employing "last write wins" strategies based on clocks or "sloppy quorum."

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

The main reason for choosing a distributed system that is not linearizable is performance. The theorem proven by Attiya and Welch, which shows that the response time for read and write requests in a linearizable system is proportional to the uncertainty in network delays. In other words, the more variable the network delay, the slower a linearizable system becomes.

Linearizability can limit availability in distributed systems, especially during network partitions. If a system requires linearizability and parts of the network become inaccessible, the disconnected parts must either wait for the network to be restored or return an error. This can result in parts of the system being unavailable to users.

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.

No, we cannot conclude that event e occurred before f simply because the Lamport clock value L(e) is less than L(f). Lamport clocks provide only a partial ordering of events, which means some events are not directly comparable.

The problem is that concurrent events (events that are not causally related) can be assigned Lamport clock values in arbitrary order. For example, if two processes p1 and p2 generate events e and f respectively, and p1 sends a message to p2 after e but before f occurs, f could end up with a higher Lamport clock value than e, even if the events occurred concurrently.

Vector clocks address this limitation by providing a more detailed view of event ordering. Instead of a single counter, vector clocks use a vector of counters—one for each process in the system. Each process updates its own counter and incorporates updates from other processes through messages.

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?

RAFT ensures a combination of mechanisms to ensure that all nodes have identical logs, even after a crash and a subsequent leader election. These mechanisms, as described in the source, include:

- Leader-based replication: RAFT has a strong leader responsible for receiving log entries from clients and replicating them to all other nodes. This ensures a unidirectional data flow and simplifies log management.
- Heartbeats and elections: Leaders send regular heartbeat messages (empty AppendEntries RPCs) to followers to maintain their authority. If a follower does not receive a heartbeat within a random timeout, it initiates a new leader election.
- Log consistency checks: When the leader sends an AppendEntries RPC to a follower, it includes the index and term number of the last entry it knows exists in the follower's log. If the follower does not have a matching entry, it rejects the new entries.
- Log backtracking: If logs are inconsistent, the leader forces the follower's log to match its own. This is achieved by identifying the last matching entry, deleting any subsequent entries in the follower's log, and sending all the leader's entries from that point onward.

- Election restrictions: To prevent a candidate from winning an election without having all committed log entries, the candidate must have a log that is at least as up-to-date as a majority of the nodes in the cluster.

8 MySQL RAFT

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

Improved Reliability and Safety:
RAFT provides "provable safety" for control plane operations such as leader (primary node) election and membership changes. This is a significant advantage compared to semi-synchronous solutions, where complex automation processes can lead to data loss in the event of a failure. By integrating the control plane and data plane into the same replicated log, RAFT eliminates the need for external automation processes, reducing the risk of errors and increasing reliability.

Simplified Operations and Maintenance:
Implementing RAFT in MySQL at Meta led to "simplified operations," as MySQL servers themselves handle leader elections and membership changes. Complex automation processes, which were previously required to ensure data consistency, became "harder to maintain over time." RAFT removes this complexity, thereby simplifying operations. Tools and monitoring systems developed to oversee the RAFT implementation helped streamline troubleshooting and reduce operational burdens.

Faster Failover Times:
RAFT provides significant improvements in failover times, achieving typical failovers "within 2 seconds" compared to "20 to 40 seconds" for semi-synchronous setups. This improvement is due to RAFT's efficient heartbeat and election mechanism, which quickly identifies and addresses leader crashes.

9 Stonebraker/Pavlo

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

Stonebraker and Pavlo argue that non-relational systems, including document databases, have either found their niche or are in the process of becoming SQL/relational model systems. They assert that SQL and the relational model remain dominant and that attempts to replace them often result in the best ideas from these alternative approaches being absorbed into SQL. In the early 2010s, a wave of document databases emerged, marketed under the "NoSQL" slogan as a response to the scalability limitations of traditional RDBMS. These databases promised better performance by avoiding SQL and JOIN operations, offering weaker transactional guarantees ("BASE" instead of "ACID").

Despite the early resistance to SQL, almost all "NoSQL" document databases now offer SQL-like interfaces. Examples include DynamoDB PartiQL, Cassandra CQL, Aerospike AQL, and Couchbase SQL++. MongoDB, which long

resisted SQL, now also offers it in its Atlas service.