Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols

Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi [*] and Xun Xue, Zhiyanan, Zhujianfeng[**]

[*] *vaibhavarora, tanujmittal, agrawal, amr*@cs.ucsb.edu, Department of Computer Science, University of California, Santa Barbara
[**] xun.xue, zhiyanan, zhujianfeng@huawei.com, Huawei

Abstract

Consensus protocols are used to provide consistency guarantees over replicated data in a distributed system, and allow a set of replicas to work together as a coherent group. Raft is a consensus protocol that is designed to be easy to understand and implement. It is equivalent to Multi-Paxos in fault-tolerance and performance. It uses a leader based approach for coordinating replication to a majority. The leader regularly informs the followers of its existence using heartbeats. All reads and writes go through the leader to ensure strong consistency. However, read-heavy workloads increase load on the leader since the followers in Raft are maintained as cold standbys. Since the algorithm itself guarantees replication to at least a majority, why not exploit this fact to serve strongly consistent reads without a leader? We propose mechanisms to use quorum reads in Raft to offload the leader and better utilize the cluster. We integrate our approach in CockroachDB, an open-source distributed SQL database which uses Raft and leader leases, to compare our proposed changes. The evaluation results with the YCSB benchmark illustrate that quorum reads result in an increased throughput of the system under read-heavy workloads, as well as lower read/write latencies.

1 Introduction

Fault-tolerance is an essential component of large-scale distributed systems. Replication is one of the most widely used techniques to achieve fault-tolerance. It is used to increase the availability of the data in case of failures. However maintaining consistent replicas across different clusters, or geographically distant datacenters, in the presence of concurrent operations, is a complex task. Consensus protocols are used to provide consistency guarantees about the data, and allow a set of replicas to work together as a coherent group.

Over the last few decades several consensus algorithms have been proposed [13, 16, 17]. Raft [17] is a consensus algorithm that is designed to be easy to understand and implement. It is equivalent to Multi-Paxos [13] in fault-tolerance and performance. It has two phases which are logically separated: leader election and log replication. A server in a Raft cluster is either a leader, a candidate, or a follower. A leader is elected in a cluster, and is responsible for replicating log to the followers. It also informs the followers of its existence by periodically sending heartbeat messages. Therefore, Raft does not need to invoke a leader election for every request. Quorum writes ensure that a majority of the cluster replicates the changes before committing. All the writes are coordinated by the leader. Since writes are not guaranteed to propagate to all the members in the cluster immediately, a strongly consistent read is performed at the leader. However, a snapshot read can be performed at any of the followers.

In Raft, all consistent read and write requests are handled only by the leader. Raft's leader-based design makes it easy to reason about the correctness of replicas, and simplifies the recovery and configuration change processes. However, this design can lead to under-utilization of resources and can affect system performance. Resource under-utilization is exacerbated in failure-free scenarios. Followers in Raft are cold replicas as long as the leader is active. The whole cluster can be better utilized if followers also can serve consistent read requests. Since a majority of members in the cluster are guaranteed to have replicated the consistent value when writing, reading from any majority in the cluster can be used to provide a consistent state. Thus, combining quorum reads with leader-based reading can help to improve read scalability in Raft. We explore such an architecture in the Raft implementation of CockroachDB [1], which is a distributed SQL database. Quorum reads help offload the load of the leader node, resulting in better utilization of followers under failure-free and read-heavy workloads. Combining quorum reads with lease-holder reads, also provides the ability to trade-off read and write latencies, by varying the fraction of reads served using a particular approach. We present the proposed approach for Raft, but it can also be applied to Raft-like consensus protocols, such as Multi-Paxos.

The remainder of the paper is organized as follows. Section 2 provides a background on consensus algorithms. We provide the implementation details of CockroachDB's Raft implementation and consistent reads in Section 3. Section 4 discusses the design to scale reads in Raft via quorum reads. The evaluation results are presented in Section 5 and we conclude in Section 6.

2 Background

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single value among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. Quorums play an important role in consensus algorithms. A quorum is the minimum number of votes that a process has to obtain in order to be allowed to make the decision on behalf of the collection of processes. Depending upon the network access strategies quorums can be as complex as $O(\sqrt{n})$ for grid-based [14], $O(\log n)$ for tree-based [6] or $O(n)$ for arbitrary accesses. Majority quorums assume equal reliability of all members of a cohort. It requires access to $(\lfloor n/2 \rfloor + 1)$ members at any given time for read as well as write operations.

El Abbadi et al. [5, 11] introduce the idea of views and Viewstamp replication [16] builds on the work, combining views with the concept of a leader (or the primary) in a given set of members (called the cohort), for replicating a state machine and recovering on failures. Paxos [12] is a general-purpose consensus protocol designed for an asynchronous environment. In Paxos, each process plays the role of a proposer, an acceptor, or a learner. Every request requires a new instance of Paxos in the system. Each request has to go through two phases to get accepted: proposal phase and acceptance phase. Consistent reads are performed using quorums (because of no stable leader). Modifications like Master-leases [8] and Multi-paxos [13] minimize electing a leader (the proposal phase in Classic Paxos) for every single request to reduce the number of rounds. This enables serving consistent reads locally at the leader. Quorum leases [15] allow any replica to acquire a lease from a majority of grantors and serve consistent reads locally. This is made possible by synchronously notifying every write to all the lease holders through the lease grantors. Other read optimizations include snapshot reads using synchronized clocks in Spanner [10] and local reads in Megastore [7].

Raft is a consensus algorithm designed with the primary goal of understandability. It separates the two concerns of leader election and log replication. Once a leader is elected, log replication is done via the leader, using majority quorums.

3 CockroachDB

3.1 Overview

CockroachDB [1] is a distributed SQL database built on top of a transactional key-value store. It replicates data over multiple nodes and guarantees consistency between replicas using Raft. Figure 1 illustrates the architecture of CockroachDB. It implements a single, monolithic sorted map from key to value. The map is composed of one or more ranges and each range is backed by data stored in RocksDB [3] (a LevelDB variant). Ranges are defined by start and end keys. They are merged and split to maintain total byte size within a globally configurable min/max size interval. Ranges are replicated to different nodes using respective consensus groups. This means that each node may be participating in multiple consensus groups with each group having an elected leader.

Leaders and Leases. Since CockroachDB uses Raft to replicate ranges, there are leaders in the cluster (one for every Raft instance). CockroachDB has an additional abstraction called leader leases. They are Raft-agnostic and are a sequence of database time intervals which are guaranteed not to overlap, for which a single replica in a Raft group has the leader lease. For this time window, the leader is assumed to be stable, hence avoiding the need to go through the expensive Raft read processing for consistent reads (i.e. waiting to hear from majority using heartbeats, after receiving the read request). Reads and writes are generally addressed to the replica holding the lease; if none does, any replica can be addressed, causing it to try to obtain the lease synchronously. The replica holding the lease is in charge of handling range-specific maintenance tasks such as splitting, merging, and re-balancing.

The lease holder can serve consistent reads locally, however, it needs to submit writes to Raft (i.e. go through the leader). The lease is completely separate from Raft leadership, and so without further efforts, Raft leadership and the leader lease might not be held by the same replica. Since it is expensive not to have these two roles co-located (the lease holder has to forward each proposal to the leader, adding costly RPC round-trips), each lease renewal or transfer also attempts to co-locate them.

Processing read/write requests. As shown in Figure 1, CockroachDB provides a SQL interface for clients. All the requests are internally treated as transactions. The system creates implicit transactions for non-transactional requests and hands them over to the Transaction Coordinator which keeps track of all the active

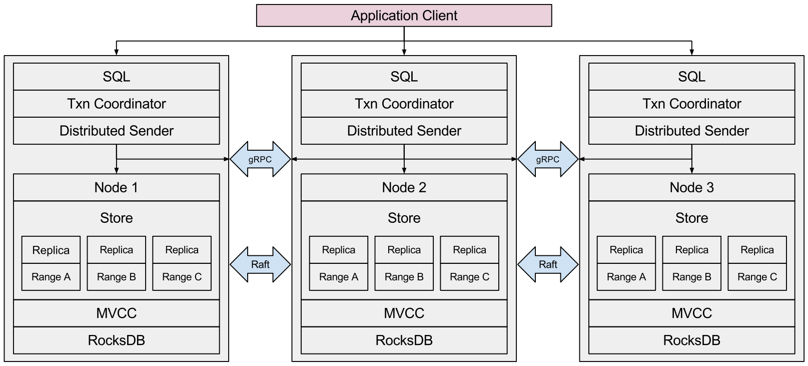


Figure 1: Architecture of CockroachDB. Shows a cluster of 3 nodes (Node 1 to Node 3), and 3 ranges (Range A to Range C) with full replication. Every range replicates using its own Raft cluster.

and aborted transactions. Application clients can send requests to any cockroachDB node. The node receiving the request is referred to as Gateway node. The Distributed Sender layer forwards requests to the respective lease holder nodes using gRPC [2]. The lease holder uses Store to retry requests in case of conflicting operations. The Replica layer proposes Raft commands for replication. The MVCC and the RocksDB layer manage the underlying data, as key-value pairs.

CockroachDB uses intents to identify conflicting write operations. Any ongoing update to a key is persisted as a write intent. It contains the proposed timestamp, value and some meta information about the transaction. These intents are cleared when a transaction completes.

3.2 Bottlenecks to read performance

As CockroachDB uses Raft for consensus, all client requests for consistent reads and writes are submitted to the lease holder. This means that irrespective of the number of replicas only the lease holder is responsible for serving reads. This could result in the lease holder getting overloaded by requests. This effect can potentially be compounded in the case of hot-spots, where only a few keys are being accessed most of the time. Range splitting can be used to reduce the effect of hot-spots, but increasing the number of ranges increases the probability of distributed transactions across multiple ranges. Hence, serving the read requests only at the range-lease holder adversely impacts the system performance as a whole. On the other hand, members other than the lease holder are just cold replicas and are not serving the client directly during failure-free executions. If we could reduce the read traffic on the lease holder by distributing the requests on other nodes then we can utilize the entire cluster more efficiently in failure-free executions and thereby sustain higher throughput.

4 Designing for Read Scalability

Raft uses write quorums before a write can be considered committed. The protocol makes sure that the updates are propagated to a majority of servers in the cluster. This property can be utilized to enable follower nodes to handle read requests. If the reads are performed on a majority of servers in the cluster, it is guaranteed that at least one of the server will have the consistent state. Therefore, read quorums can be used to serve strongly consistent reads from the followers. Two variants of quorum reads are proposed. A simple majority read provides an efficient solution, but has a pathological case where the value returned might not be the latest linearizable read. This approach provides the advantage of an efficient read, but has a low probability of returning a stale value. The second variant of the quorum read returns the strongly consistent value corresponding to a key, but can be more expensive.

4.1 Quorum Read Approaches

Quorum Reads In the basic approach, the gateway node (the node receiving the client request) sends the read request to a majority of the servers. Every node replies with a timestamp along with the data, corresponding to the latest stable value at the node for the key read. Since a committed value must be present in one of the servers among any majority, the value corresponding to the highest timestamp is the latest committed value. This value is chosen and sent back to the client. The pathological case for this approach is that a particular value might be in the process of committing; a majority of servers might have responded to the leader to agree to the write request, but might not have heard back from the leader yet. Hence, the leader might commit ahead of the value read using the quorum read approach. This approach also provides a trade-off to a local read, as it is more expensive than reading the local value at a replica, but has a much higher probability of returning the latest value.

Strongly Consistent Quorum Reads To overcome the pathological case of the simple quorum read approach, we also propose and implement a strongly consistent quorum read. As described earlier, Cockroach DB uses write intents for recording the proposed values at a server. When a node receives a request for a strongly consistent quorum read, in the case of conflicting write intents for a particular key, the node replies with only the timestamp, but no data. Sending a timestamp and no corresponding data signals that a write intent was encountered for that particular key. As the read is sent to a majority of servers, any possible ongoing write request that could have been committed at the leader will be detected. This is due to the fact that for the leader to commit a value in Raft, a majority must have appended the request to their respective copy of the log. The gateway node then selects the data with the latest timestamp. If the data is available, it implies that the timestamp corresponds to the latest stable value. However, if there is no data available, corresponding to the highest timestamp, it implies that there are pending updates in the system and the request is considered as failed. A back-off policy is used for subsequent retries.

4.2 Combining Lease-holder and Quorum Reads

As write requests still go through the lease-holder, the latest value can be read from the lease-holder like before. Our proposed approach also combines the lease-holder based and quorum reads. If the gateway node is the lease holder, it can retrieve consistent state locally. However other nodes have two choices:

1. Read from the lease holder, or
2. Read from a majority (excluding lease holder)

To read from a majority, a basic approach could be to send requests to all the servers in the cluster (except the lease holder) and wait to hear back from a majority before replying to the client. This approach would work but could end up generating a significant load on all the servers. Instead we can use a random selection approach to send requests to only selected servers in the cluster to form a majority. We make this choice to optimize for failure-free executions.

A read request is executed either as a quorum read or a lease-holder read, such that the read requests are uniformly distributed over all the nodes. Assuming that a cluster of n nodes is fully replicated, every node gets equal number of client requests, and a gateway node always includes itself for majority, the read request to the non-lease holder nodes uses the lease-holder read for $x\%$ of total reads and uses quorum reads for all the other requests. Solving for x for distributing the read requests uniformly in the cluster, we get,

$$x = \frac{P*(n-2)}{n+P*(n-2)} \times 100$$

where P is probability of a non lease-holder node being included in a majority quorum by other non-lease holder nodes

$$P = \begin{cases} 1 & n = 3 \\ \dfrac{\binom{n-3}{\lfloor n/2 \rfloor - 1}}{\binom{n-2}{\lfloor n/2 \rfloor}} & n > 3 \end{cases}$$

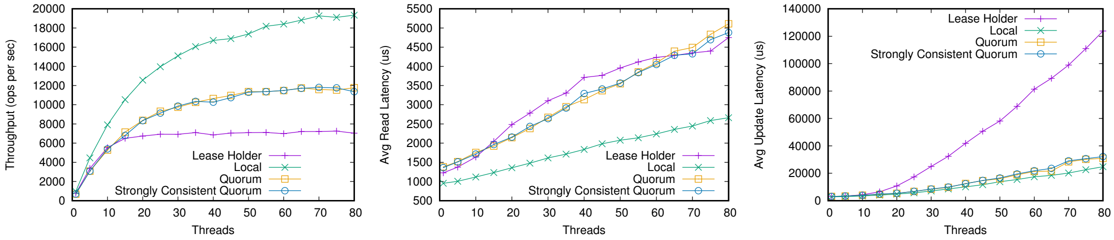
Using the combination of lease holder reads and quorum reads we can guarantee consistent reads while not overloading the lease holder. Also, all the cluster members would be utilized for serving read requests and the lease holder would not be a bottleneck. The fraction of reads being served using quorum reads can be configured based on the read and write latency trade-off desired. Having a higher fraction of reads served by the non-lease holder nodes can help reduce the load on the lease-holder and reduce write latencies, at the expense of read latencies and increasing the load of non-leader nodes.

5 Evaluation

YCSB [4, 9] is used to benchmark and analyze the performance of the proposed approach of combining quorum reads with traditional lease holder reads in CockroachDB's Raft implementation. We compare four approaches in the evaluation: *Lease-holder reads* (default baseline approach of reading from the lease holder), *Local reads* (read the local value at the replica), *Quorum reads* (read latest value from a majority), and *Strongly consistent quorum reads* (quorum reads considering ongoing write requests). Local reads may return inconsistent results, but provide a measure of the upper bound of read performance. Both the quorum read and the strongly consistent quorum read approaches also perform a fraction of reads at the lease holder to ensure equal distribution of read requests, as described in Section 4.2.

We use a fully replicated cluster of 5 AWS EC2 machines (m3.2xlarge instance type), with range size big enough to avoid any range splits during the experiments. The fraction of lease-holder reads was set to 28.57%, based on the calculation of x (Section 4.2). Another machine was used as YCSB client to generate equal load on all the nodes. The dataset comprises 100K records with each record having a key and a value. Unless otherwise mentioned, the workload is read-heavy, with 95% reads and 5% writes, and is uniformly distributed over the keys.

Scaling Up Client Threads. Figure 2 illustrates the performance of all the approaches under varying number of client threads. We observe that both the quorum read and strongly consistent quorum read achieve higher throughout than the traditional lease holder read approach (around 60% higher), where all the reads are



(a) Throughput (b) Read Latency (c) Write Latency

Figure 2: Scaling-Up Clients - Uniform Distribution

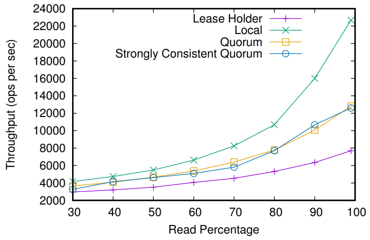


Figure 3: Varying % of reads - 70 client threads

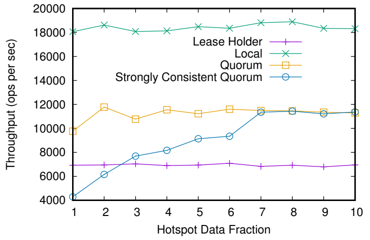


Figure 4: Hotspots - 70 client threads

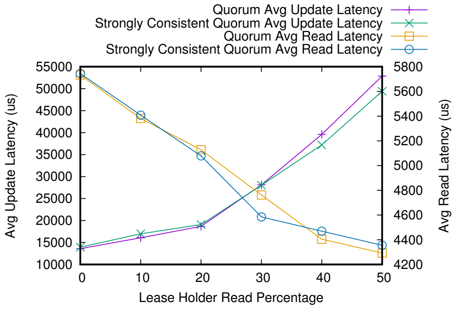


Figure 5: Varying Fraction of Lease-Holder Reads - 70 client threads

performed at the lease holder. Figure 2c also shows that distributing the reads is highly beneficial in mitigating the increase in write latency, as the load on the system increases. As both the quorum read approaches reduce the load on the lease holder, around 4x improvement in write latency is observed with 80 concurrent client threads. Both the proposed approaches achieve a lower read latency with more than 10 client threads, due to the distribution of read requests. But as the load on the system increases, the higher number of reads involved in the quorum reads result in reducing the gap in read latencies.

Varying Read-Write Ratio. In Figure 3, we observe that as the percentage of reads increases from 30 to 99, the throughput gap between the quorum read approaches and the traditional lease holder read approach increases. This is due to the fact that increasing fraction of reads can be efficiently distributed among the cluster via quorum reads, resulting in better throughput.

HotSpots. Figure 4 illustrates the performance under hotspots, where 80% of the requests access a restricted fraction of the data (specified in % of the entire data on the horizontal axis). We observe that under really high contention (left side of the horizontal axis), the throughput achieved by the strongly consistent quorum read approach is much lower than quorum reads, because of the higher probability of accessing a key within an ongoing write request. While, as the contention decreases, the throughput gap between the two proposed quorum read approaches decreases. For other approaches, the throughout does not vary a lot due to the low percentage of write requests.

Read-Write Latency Trade-Off. In the proposed quorum read approaches, the fraction of reads to be performed at the lease-holder provides a way to trade-off read and write latencies. Figure 5 illustrates this trade-off. As we increase the fraction of reads performed at the lease holder, the write latency increases due to the increase in load on the lease holder. On the other hand, we also observe that serving more reads from the lease holder at a high load, reduces the read latency. The highest throughput was achieved at 20% lease holder reads, when the load is almost uniformly distributed throughout the cluster. Combining the quorum read approaches with the traditional lease holder reads, provides a mechanism to trade-off between read and write latencies.

6 Conclusion

We propose combining quorum reads, with traditional single leader based reads in Raft-like consensus protocols. A basic quorum read, which can return non-linearizable value in a corner case and a strongly consistent quorum read approach are proposed. Results with YCSB benchmark demonstrate that the proposed approach results in higher throughput and improved read/write latencies with read-heavy workloads, and better utilization of the follower nodes in failure-free scenarios.

7 Discussion Topics

Leader-based consensus protocols like Raft and Multi-Paxos rely on leaders for both read and write requests. This simplifies operations and recovery, but generates most of the load on leaders whereas followers only act as cold standbys. Using read quorums can help distribute the load from leader to its followers. This results in higher throughput for read-heavy workloads.

An interesting topic of discussion would be about the read and write latency trade-off of combining the lease-holder reads, with the quorum reads. Traditional data management systems trade-off throughput and latency. The proposed approach allows to fine tune between read and write latencies, and can be configured based on the application requirements. Another topic which we feel can spark debate is the corner case in the quorum read approach, where a leader might have an ongoing commit, while the basic quorum read approach is in progress. Identifying applications which can or cannot tolerate such reads, is an interesting point of discussion.

An area of future work is how to choose a majority of nodes for quorum reads. In the current work, all the non-lease holder nodes are chosen uniformly. In the future, these decisions can be based on varying metrics such as server's resource utilization (CPU, memory and disk), cluster's network traffic and latency from different nodes in the cluster. All the servers in the cluster can share such metrics periodically using a gossip protocol. The current choice of majority of servers is optimized for failure-free scenarios as well. It would be interesting to look at performing quorum reads during failures.

The proposed approach is more suited to read-heavy workloads. With the increase in the ratio of writes, the benefit of the quorum approaches decreases. Furthermore, as the data contention and the write ratio increase, strongly consistent quorum approach might lead to more retries, and the basic quorum read approach has a higher probability of returning a value which is one commit behind the leader.

Acknowledgements: This work is supported by a grant from Huawei.

References

[1] CockroachDB. `https://www.cockroachlabs.com/`.

[2] gRPC: A high performance, open-source universal RPC framework. `http://www.grpc.io/`.

[3] RocksDB: A Persistent key-value store for fast storage environments. `http://rocksdb.org/`.

[4] YCSB: Yahoo! Cloud System Benchmark. `https://github.com/brianfrankcooper/YCSB`.

[5] Abbadi, A. E., and Toueg, S. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems (TODS) 14*, 2 (1989), 264–290.

[6] Agrawal, D., and El Abbadi, A. The tree quorum protocol: An efficient approach for managing replicated data.

[7] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011), vol. 11, pp. 223–234.

[8] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (2007), ACM, pp. 398–407.

[9] Cooper, B. F., Silberstein, A., Tam, E., et al. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC* (2010), pp. 143–154.

[10] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[11] El Abbadi, A., Skeen, D., and Cristian, F. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems* (1985), ACM, pp. 215–229.

[12] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS) 16*, 2 (1998), 133–169.

[13] Lamport, L., et al. Paxos made simple.

[14] Maekawa, M. An algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS) 3*, 2 (1985), 145–159.

[15] Moraru, I., Andersen, D. G., and Kaminsky, M. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–13.

[16] Oki, B. M., and Liskov, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (1988), ACM, pp. 8–17.

[17] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), pp. 305–319.