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
Distributed Systems Lab (DSL)
University of California, Santa Barbara
vaibhavarora@cs.ucsb.edu

Xun Xue, Zhiyanan, Zhujianfeng
Huawei
Large-scale distributed systems are now ubiquitous

Advent of the cloud have made them more accessible

Failures are now the norm, and have to be dealt with
Replication and Consensus

- Large-scale distributed systems need to be fault tolerant

- **Replication** is a technique to achieve fault tolerance

- Replication brings in added complexity in synchronizing multiple data copies

- **Consensus Protocols**
  - Allows set of Replicas to act as a coherent group
  - Goal is to have multiple processes agree on a common value
  - **Quorums** – Minimum number of votes to make a decision for a collection of processes
Consensus Protocols

- **Paxos** and variants
  - Classic Paxos, Multi Paxos, Fast Paxos
  - Widely used in recent large-scale distributed systems
    - GFS, Megastore, Spanner, Ceph etc

- **Raft**
  - Designed with the goal of understandability
  - Separates Leader election and Log replication
Many applications need **Linearizable** reads.

- Our industrial partners, Huawei, have these demands too
- Consensus protocols can help provide these guarantees

Variants for read-optimized settings

- **Master Leases** – Multi-Paxos
- **Quorum Leases** – SOCC 2014
- **Read-Optimization in Megastore** – Read-any, write all
Raft

Leader Election

Log Replication

- Leader proposes a value to the cluster
- Followers accept the proposal and reply
- Leader waits to hear from a majority, commits the value locally and notifies the cluster
- Followers also commit the value

Linearizable reads at the leader – wait a round of heartbeats
CockroachDB

- An open-source, fault-tolerant, strongly consistent, scale-out SQL database

- Inspired by Spanner

- Storage
  - Data sorted as single monolithic key-value map
  - Divided into partitions / ranges replicated by Raft

- Lease-Holder – Non-overlapping leases
Logical Overview of CockroachDB

SQL

Distributed, Monolithic KV Store

Node 1
- Range A
- Range C
- Range D

Node 2
- Range A
- Range B
- Range D

Node 3
- Range A
- Range B
- Range C

Node 4
- Range B
- Range C
- Range D
Journey of a Request

1. Client request
2. Gateway Node
3. Lease Holder
4.  
5.  

Diagram showing the journey of a request from the client to the gateway node and finally to the lease holder.
1. Forward the request

2. Creates write intent, and replicates using Raft

3. Replication successful at a majority

4. Mark transaction successful, clear intents, and return success
1. Forward the request

2. Return the response

3. Complete client request

If there is write intent, based on priority:
- Abort
- Wait until intent is cleared

4.

5.

Gateway Node

Lease Holder

Read Request
Bottleneck to Read Performance

- Reads are executed at the Lease-holder

- Overloads Lease-holder
  - Can be reduced by partition / range splitting – but this has many challenges - percentage of distributed transactions across ranges increases, find the right partitioning strategy is hard, hotspot partitions will still cause read bottlenecks
  - Followers are cold standbys during failure-free scenarios

- Can we use the follower nodes for Linearizable reads? And optimize for read-heavy workloads?
Improving Read Scalability

- Raft uses Majority Quorums to **commit writes**
- We exploit this fact to read from a **majority quorum**
- Combine with Lease-holder reads
Quorum Reads

- Send read requests to a majority of nodes
- Every node replies with latest stable value with corresponding timestamp
- Choose the value with latest timestamp
Quorum Reads

1. Client read request
2. Select random majority, read values and timestamps
3. Nodes reply with timestamp and value
4. Selects T* and corresponding value V*
5. Return V*

T* > Ts

Gateway Node

Lease Holder

K : V*, T*

K : V, Ts

K : V, Ts

K : V, Ts

K : V*, T*

K : V*, T*
What if there is an ongoing request committed at the Lease-holder?

**Strongly Quorum Reads**
- Use Write intents to detect ongoing writes
- In case of conflicting writes, every node replies with timestamp and no value
- At gateway node, if there’s no value corresponding to latest timestamp, retry with a backoff

This approach can serve linearizable / strongly consistent reads
Strongly Consistent Quorum Reads

1. Client read request

2. Select random majority, read values and timestamps (considering write intents)

3. Nodes reply with timestamp and value. If there is a write intent, value is send as null.

\[ T^* > T \]

Gateway Node

1. Client read request

3. Selects \( T^* \), no corresponding value found. FAILED attempt. Retry.

5. \( K : V, T^s \)

4. \( K : V, T^s \)

3. \( K : V^*, T^s \)

17
**Strongly Consistent Quorum Reads**

1. **Client read request**

2. **Gateway Node**
   - Selects $T^* \text{ and corresponding value } V^*$
   - Selects $T^* \text{ and } V^*$, $Ts$

3. **Lease Holder**
   - $K : V^*, T^*$

4. **Node 4**
   - $K : V^*, T^*$

5. **Node 5**
   - $K : V, Ts$
   - $V^*, T^*$

6. **Nodes reply with timestamp and value. If there is a write intent, value is send as null.**
   - 6. $null, T^*$

7. **Selects $T^*$ and corresponding value $V^*$**

8. **Return $V^*$**
   - $V^*, T^*$
   - $V, Ts$

- $T^* > Ts$

- **Write Intent might resolve before retry**
Combining Lease-holder Reads and Quorum Reads

- Lease-holder can always read from local store
- Non lease-holders can read from:
  - Lease-holder, or
  - Majority
- To uniformly distribute read requests over all nodes, assuming:
  - a cluster of \( n \) fully replicated nodes
  - every node gets equal no. of read requests
  - a node always includes itself for majority

A gateway node can use lease-holder for \( x\% \) of total reads, and quorums for others

\[
x = \frac{P(n-2)}{n+P(n-2)} \times 100
\]

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

\[
P = \begin{cases} 
1 & n = 3 \\
\frac{n-3}{\lfloor n/2 \rfloor - 1} & n > 3
\end{cases}
\]

Provides ability to trade-off read & write latencies
The proposed approaches are integrated within CockroachDB. Available on GitHub. 
https://github.com/vaibhavarora/cockroach/tree/raft-read-scalability

YCSB Workload. Dataset of 100K items with (key, value)

CockroachDB cluster of 5 AWS EC2 machines (m3.2xlarge instance type). 1 machine for YCSB clients

4 different read strategies
- Lease-holder reads
- Local reads – an upper bound on performance
- Quorum reads
- Strongly consistent Quorum reads

Uniform read distribution throughout the cluster – 28% lease-holder reads for both proposed quorum read approaches
Scaling Clients

Uniform workload (95% reads, 5% writes)

Improvements with Quorum reads:

- ~4x write latency
- ~60% throughput
Varying Read-Write Ratio

- Uniform workload
- **Varying read requests** (30% to 99%)
- 70 client threads

**Higher the read %, higher** is the benefit of using the quorum read approaches

Up to **~85% improvement** in throughput using Quorum read approaches
HotSpot workload - 80% requests access varying data (1% to 10%) (95% reads, 5% writes)

At high contention, strongly consistent quorum reads have a large number of retries because of frequent conflicts.
Read-write latency tradeoff

Varying lease-holder reads (0% to 50%)  
Uniform workload (95% reads, 5% writes)  
70 client threads

• Quorum read approaches reduce load on lease-holder, leading to improved write latencies

• Lease-holder reads reduce read latency

Read and write latencies curves Intersect near the point of Uniform read distribution
Future Considerations / Discussion

- Can we choose majority in a more intelligent way?
  - Use resource utilization & network latencies

- How well can quorum reads perform in failure-prone scenarios?

- Look into using strongly consistent quorum reads as part of transactional mechanisms

- Further improving read latencies – maybe for a subset of keys
Conclusion

- Proposed **Quorum read** approaches for Raft-like consensus protocols
- **Combine** them with traditional lease-holder reads
- Provide a way to trade-off between **read & write latencies**
- For failure-free scenarios with read-heavy workloads:
  - Improved throughput
  - Highly Improved **write latencies**

vaibhavarora@cs.ucsb.edu