Linearizable Quorum Reads in Paxos

Aleksey Charapko[1,2], Ailidani Ailijiang[2], and Murat Demirbas[1,2]

[1]University at Buffalo, SUNY
{acharapk, demirbas}@buffalo.edu
[2]Microsoft, Redmond, WA
{ailidani.ailijiang}@microsoft.com

Abstract

Many distributed systems/databases rely on Paxos for providing linearizable reads. Linearizable reads in Paxos are achieved either through running a full read round with followers, or via reading from a stable leader which holds leases on followers. We introduce a third method for performing linearizable reads by eschewing the leader and only reading from a quorum of followers. For guaranteeing linearizability, a bare quorum read is insufficient and it needs to be amended with a rinse phase to account for pending update operations. We present our Paxos Quorum Read (PQR) protocol that implements this. Our evaluations show that PQR significantly improves throughput compared to the other methods. The evaluations also show that PQR achieves comparable latency to the read from stable Paxos leader optimization.

1 Introduction

The Paxos consensus protocol [15] is used for implementing a fault-tolerant replicated state machine (RSM), where each node executes commands in the same order and maintain identical states. Many distributed systems use Paxos as the backbone for strongly consistent distributed replication [7, 8, 10, 12, 26]. To guarantee strong constraints on the safe interleaving of updates and provide ease of reasoning/development, these distributed systems often provide linearizability to prohibit stale reads: once an operation is complete, every read must see that state or some later state. In other words, every read sees some current state between invocation and completion, but not a state prior to the read.

A straightforward way to serve a linearizable read operation in Paxos is to run this command through a majority of followers. But this is a costly solution because executing a Paxos round for a command that does not change state is wasteful. Furthermore, read operations occur more frequently than write operations, and read-to-write ratios greater than 100:1 are not uncommon in the industry [21].

To reduce the cost of linearizable read operations, read from the leader optimization is performed [9, 20]. This approach requires the leader to be stable. To assert this condition, the leader employs leases with the follower nodes which prevents the election of another leader during the lease period.

Both of these methods involve the leader, which is already burdened by serving every write operation [2]. Making the leader responsible for all read traffic puts even more strain on it, while the follower replicas perform no client-facing work and exist solely for the purpose of fault-tolerance and redundancy.

Contributions. We present a general client-driven, strongly consistent quorum read protocol, called Paxos Quorum Reads (PQR). PQR relieves the Paxos leader from serving read operations and achieves more uniform resource utilization among the nodes in the system. PQR also offloads the burden of orchestrating the read to the client, and provides better scalability for high throughput read/write workloads in Paxos-based databases.

Providing linearizable quorum reads in Paxos is not an easy problem. The naive approach of simply reading the committed value from the majority of followers may result in stale reads and read-your-write violations: A client that receives an ack for a successful write may not see its own write upon a subsequent read from a quorum of followers. This is because the followers do not have the most up-to-date information about the committed writes. The followers know only about the *dirty* writes that they "accept", but not about the "commit" finalized by the leader upon seeing a quorum of accepts.

In order to avoid this problem in PQR, after reading accepted values from a quorum of follower replicas, the client is then required to rinse a dirty read by waiting for the highest "accepted" slot to become "committed". While this solution might appear slower than a point-read from the leader node, and not very advantageous at a first glance, it provides value by alleviating the need for leader-leases and by ensuring a more even load distribution among the nodes in the cluster for intensive write and read workloads. Moreover, as we show in Section 4.1, the rinse phase becomes unnecessary and is omitted in many cases.

Our prototype implementation of PQR achieves up to 12%

better throughput and up to 30% better latency in write intensive workloads, while having performance similar to Paxos with reads from leader in read-intensive scenarios.

2 Related Work

Partly due to the bottleneck the linearizable reads create on the leader, many practical systems delegate the reading of data to ad-hoc protocols build on top of the Paxos-backed replication layer. Many systems, such as ZooKeeper [5], resort to reading data from any replica, which may return stale data.

A prominent group of replication protocols used in databases [11] forgo Paxos and similar, and use client-driven quorum systems [19] to implement both reads and writes. Traditional quorum systems, used in databases like Cassandra [11], utilize simple nodes with no knowledge of global state. Once a replication operation reaches a storage node, it is applied and committed right away. As a result, the storage node in these systems lack global knowledge of operation state: a local commit does not guarantee the success of operation globally. This prevents Cassandra from achieving linearizability without employing a costly read-repair phase: before returning the read value, that value should be written to and acknowledged by a majority replicas to ascertain linearizability/durability.

To account for the lack of knowledge about the state of global replication, some quorum protocols employ logical sequence numbers (LSNs). LSNs, however, only work when there is a single stable source to generate them. In practice, this means that these quorum systems need to rely on leases and orthogonal leader-election mechanisms, often implemented with Paxos, to ensure only a single active LSN-generating leader exists. Examples of this approach has been demonstrated in Chain Replication protocol [24], CRAQ [22], and Distributed Versioning [3] work. In Amazon Aurora [25], a stable leader must successfully write to a majority of storage replicas before an operation is considered committed. Upon reading, the stable leader knows that commitment status and may return the read by reading from any replica that have the value stored. In case of the leader failure, however, the commitment information stored at the leader is lost, and the new leader must recover it by reading the storage replicas. In contrast to the above work, that are constrained to a single datacenter, Azure Cosmos DB [17], by virtue of a novel nested consensus protocol, allows strong reads on secondaries using a proprietary protocol even in globally-distributed deployments.

In Paxos Store [26], a follower contacted for a read, queries majority of followers and responds with the read value if there is no pending write operation (i.e., dirty write) in the system. Paxos Quorum Lease protocol [18] employs leases at followers to provides local reads at those followers when there are no dirty writes. The lease is integrated to Paxos write operation and is per object. Using leases leads to a complicated implementation due to lease set, renewal, and revocation messages, and brief unavailability problems for read/write operations in case a follower holding a lease becomes unavailable.

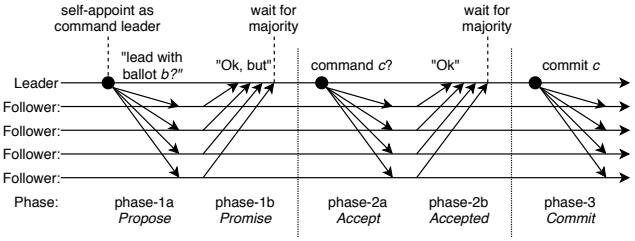

Figure 1: The three phases of Paxos protocol

Another quorum read solution [6] has been proposed recently for CockroachDB [9], a database which uses Raft for replication. The solution relies on the CockroachDB's transaction layer, orthogonal to Raft, to provide stronger consistency guarantees than the naive implementation. CockroachDB relies on HLC [14] timestamps to order transactions across different Raft-groups, and the proposed protocol piggybacks on the HLC timestamp to provide the ordering heuristic. Since Cockroach DB is not linearizable (or strictly-serializable for transactions) due to the reliance on timestamps originating at different servers, the quorum read protocol taking advantage of these timestamp is not linearizable either and provides only serializability.

In contrast to previous work, PQR leverages on Paxos slots (i.e., consensus instances) as LSNs, but also uses global replication state (communicated via "committed" messages) to reduce the number of follower reads to confirm linearizability. Unlike quorum systems relying on contiguous LSNs, PQR can tolerate temporary gaps in slot numbers, as these gaps can happen normally in Multi-Paxos, and nodes rely on the logs to identify the gaps and recover as needed. PQR does not use leases, and works efficiently and consistently in the presence of a dirty write by waiting to see the completion of the write.

3 Paxos Background

The Paxos protocol operates in three phases as illustrated in Figure 1. The first phase, often called propose phase, establishes a leader node. In this phase, a node tries to acquire leadership by reaching out to other nodes with some ballot number. The replicas ack the leadership proposal only if they have not seen a higher ballot. When a node collects a majority of acks, it can proceed to the second phase. In phase-2, the accept phase, the leader tells all the followers to accept a command. The command is either a new command of the leader's choosing, or an old command if some node has replied with an earlier uncommitted command during the phase-1 vote. Once the leader receives a majority of acks from nodes accepting the command, which anchors the command, it can proceed to the commit phase (phase-3). During this phase, the leader

instructs the followers to finalize the command and commit it to the log and execute it against their state machines.

This basic Paxos protocol is commonly extended to the Multi-Paxos [23] protocol, which allows the same leader to propose commands in subsequent slots (i.e., consensus instances), alleviating the need for leader re-election in phase-1 as long as the incumbent leader's ballot number remains the highest the followers have seen. Additionally, phase-3 is often piggybacked to the next phase-2, further reducing the communication overheads. In this paper we refer to the Multi-Paxos optimization when we mention Paxos protocol.

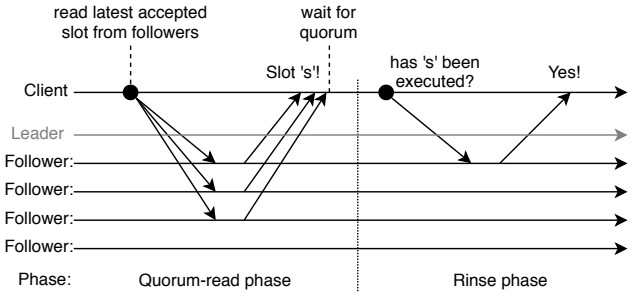

Figure 2: Overview of PQR algorithm

4 Paxos Quorum Reads

As shown in Figure 2, our Paxos Quorum Read (PQR) protocol operates in two phases: quorum-read phase and rinse phase.

Quorum-read phase. In the quorum-read phase, PQR client reads the latest accepted slot from the majority of followers excluding the leader node. As each follower replies to the client, it returns the latest accepted slot seen by that follower regardless of the gaps in the slot numbers. For example, if a replica has accepted slots 1 through 4 and 6, it will return slot 6 as its latest one. The client collects the responses from a majority quorum of followers and picks the highest accepted slot number for the rinse phase. The result from this slot will be eventually returned as the outcome of the read operation, however, at this point the operation has not been committed yet and requires some waiting to ensure its value is permanently written. This guarantees that clients always read the latest value with respect to the real-time ordering of events.

Rinse phase. In this phase, the client waits for the slot number acquired in quorum-read phase to be executed. To this end, the client needs to contact any single node and check whether the command at this slot has been executed. It is sufficient to check only a single follower for execution confirmation, since the executed state carries information about global replication in the system: the command executes only when all commands on prior slots have been committed in the cluster. If the current slot has been executed, the client can complete the read operation and use the result of that slot's executed command. Else, if the command is still pending, the client must retry the rinse phase again.

4.1 Rinse-free Reads

Following quorum-read with a rinse phase requires at least two RTTs to complete. However, in practice it is possible to eliminate the rinse phase in many cases.

Firstly, in the absence of new commands, the followers are fully caught-up with their latest accepted slot equal to their latest executed slot. When the quorum-read phase meets this condition, we can accept the read value immediately without having to perform the rinse phase, since the rinse condition has been met in just one RTT.

Secondly, when Paxos is used in databases operating on multiple data objects, such as micro-shards [4] or key-value pairs, it is possible to extend the above observation. If each replicating value has a unique identifier or key, we can track the replication progress of individual data objects. This allows PQR to compare accepted and executed slots of individual objects. If accepted and executed slots for an object match on the majority of nodes, we can safely return the value, since no replication operation for that specific object was able to reach the majority nodes at the time a PQR read was started. However, if at least one replica contacted in quorum-read phase has different accepted and executed slots for an object, then the replication is currently in progress, and the client must perform the rinse phase on this slot.

4.2 Flexible Quorum Optimization

We further improve the performance of the quorum-read phase by leveraging on the flexible quorums idea [13]. This result shows that we can weaken the "all quorums should intersect" assertion in Paxos to instead "only quorums from different phases should intersect". That is, majority quorums are not necessary for Paxos, provided that phase-1 quorums (Q_1) intersect with phase-2 quorums (Q_2). This allows trading off Q_1 and Q_2 sizes to improve performance. For high read-to-write ratios, it is beneficial to use a large $Q2$, which allows us to use a smaller than majority quorum for PQR. For example for deployment size $N = 5$, selecting $Q2 = 4$ enables us to use 2 nodes for the quorum-read phase, as this quorum is guaranteed to intersect with the write quorum $Q2$. Similarly, for $N = 7$, it is possible to choose $Q2$ as 5, to allow using 3 nodes for the PQR quorum. This provides savings for high read-to-write ratio deployments as the PQR quorum is exercised more than the write quorum. Moreover, this also helps for achieving reads from a small number of nodes within the vicinity for wide area network deployments of Paxos.

Table 1: Number of messages handled by each node, where n denotes number of nodes and r the read probability

Component	Paxos	Paxos Read From Leader	Paxos Quorum Read
Leader	$2n$	$2n(1-r)+2r$	$2n(1-r)$
Follower	2	$2(1-r)$	2
Client	2	2	$2(1-r)+2r(\lfloor \frac{n}{2} \rfloor + 1)$

5 Communication Complexity Analysis

We analyze communication complexity of Paxos and our proposed PQR solution to illustrate that PQR may achieve better performance than Paxos. To this end we consider the number of messages handled by Paxos leader, followers, and clients. Table 1 illustrates how many messages a stable leader, followers and clients need to process for each operation in a cluster of size n. We distinguish two types of operations: reads and writes, with r designating the probability of reads.

The best case for communication in Paxos is when a stable leader is known to the clients. The leader runs the client's request by communicating with the followers and committing the operation on the log. Being a general solution, Paxos does not distinguish between operation mutating the state and the one reading the state. We also assume that a leader only needs one phase of communication to commit the data, with phase-3 of Paxos piggybacked to the next phase-2. This results in 2n messages handled by the leader and 2 messages handled for the client and each follower per consensus instance.

When optimizing Paxos for read-intensive applications such as databases, in order to reduce the read communication overheads, read operations are performed directly at the stable leader [9], which holds leases on followers. In this case, read and write operations have different costs. The followers are only involved in write commands, so each follower handles 2*(1-r) messages per slot on average. The leader handles writes which it clears through the followers—resulting in 2n*(1-r) messages—, and serves reads locally—resulting in 2r messages.

PQR does not change the write path on Paxos, but performs reads from a quorum of follower nodes without contacting the leader. Most often the data can be read in just one RTT between a client and followers, however in some cases more rounds may be necessary if read conflicts with an ongoing write. In our analysis we present the best case (lower bound) on communication costs in PQR, and as we show in Section 5, PQR rarely requires more than one RTT. This means the leader handles $2n(1-r)$ messages and each follower only 2 messages per slot on average. The client, on the other hand, sees an increased workload as it coordinates the PQR reads with the followers.

Our communication comparison shows that read from leader Paxos extension reduces the message load on both the leader and followers compared to the standard Paxos. PQR reduces the message load at the leader even further. Moreover,

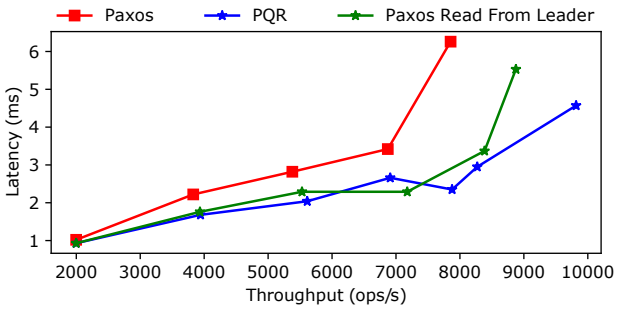

Figure 3: PQR throughput and latency

PQR offloads work outside of the Paxos cluster to the clients, and makes the clients responsible for coordinating the read operations.

6 Evaluation

We implemented and evaluated Paxos Quorum Read (PQR) protocol in Go using Paxi [1] framework for prototyping and evaluation of consensus and replication protocols. PQR required minimal changes to the Paxi's Paxos algorithm and it can be enabled or disabled through the framework's configuration. However, Since PQR is a client-driven protocol, we had to modify Paxi's client code.

We evaluate our prototype PQR on AWS EC2, with Paxos and PQR replicas deployed over 5 m5a.large instances with 2 vCPUs and 8 GB of memory. We ran the Paxi benchmark on a single m5a.xlarge instance with 4 vCPUs and 16 GB of RAM. Our microbenchmark generates a workload for a key-value store containing 1000 distinct objects. Each operation either updates or reads a small 8-byte value. We ran each test for 2 minutes and averaged the latency accrued over that interval.

First, we compare the throughput and latency of PQR implementation against Paxos and Paxos with read from leader optimization. We use a heavy write workload with 75% of all operations being writes. Figure 3 shows that PQR performance is comparable to the Paxos with read-from-leader optimization. In this workload PQR shows better latency and about 12% higher maximum throughput. We also conducted similar experiments at 25% and 50% write ratios with similar results, although the advantage of the PQR at lower write ratio diminishes and maximum throughput become comparable with that of Paxos reading from leader mode. This is because write replication no longer contributes significantly to the load at the leader, allowing it to serve reads as efficiently as the followers in PQR.

Figure 4 shows the latency of PQR and Paxos read from leader optimization at 4000 operations per second for different write ratios. PQR exhibits slightly lower latency at workloads with higher write ratios, due to the more even load distribution in the cluster, as the read operations do not compete for

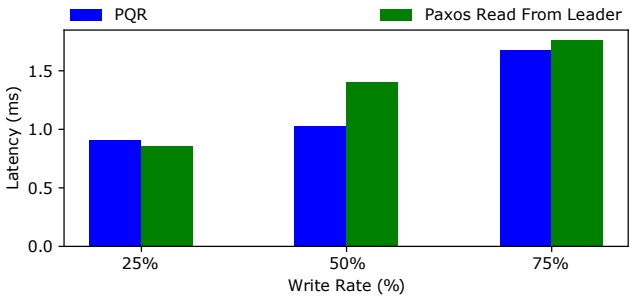

Figure 4: PQR latency at different write ratios

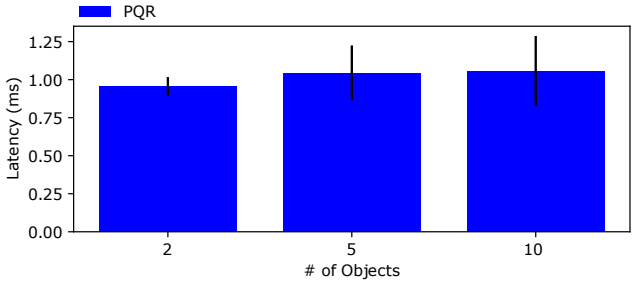

Figure 5: PQR latency with a high contention workload

resources on the leader node with writes.

Figure 5 shows PQR latency in a high contention experiment using a small number of objects as hot-spots. In this experiment, we ran a workload with 2, 5, and 10 keys at 4000 requests per second and 50% write ratio. High object contention, in theory, causes PQR to go into the rinse-phase and increases the latency of a read round, however, in practice we find that this effect is not significant. For example, when we have just 2 objects with 4000 ops/sec, with 75% write workload, PQR experiences around only 155 rinse-phases per second, causing these reads to take one additional RTT. In our workload, no read operation took more than 2 RTTs to complete. As shown in Figure 5, the impact of high object contention on latency is not easily measurable. In fact, the 2-key high contention workload actually performed slightly better (but well within the standard error) than the 5 and 10 key workloads.

We used the linearizability checker in our Paxi framework to confirm that PQR's execution is linearizable. The checker, based on the algorithm in [16], has not shown any violations in over a hundred iterations of 30-second test runs of the protocol under a variety of workloads and client concurrency.

In order to investigate the number of linearizability violations that occurs in the absence of a rinse phase, we performed experiments with reading from any single node and reading from a quorum with the rinse phase disabled. We performed the experiments in one machine using 10 keys with 50% write workload by increasing the number of clients to create contention. Figure 6 shows that reading from the quorum reduces

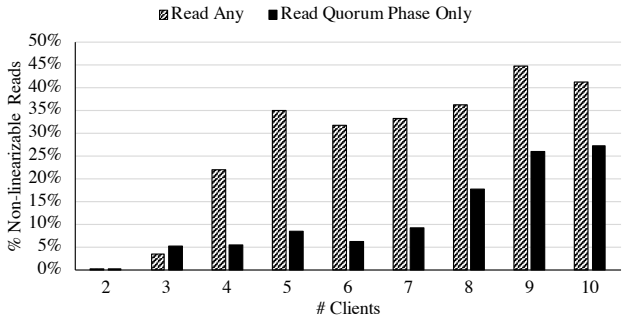

Figure 6: Linearizability violations in the absence of a rinse phase

the number of non-linearizable reads compared to reading from any single node. The single node read may miss the latest confirmed write, as it may miss the write quorum that accepted it or the leader that confirmed it. By reading from more number of nodes, the quorum read guarantees reading at least the latest anchored write value. However, it may also return newer unanchored writes, which may then be missed by the next quorum read operation resulting in a linearizability violation. The number of non-linearizable reads is still significant in either case, and this emphasizes the importance of the rinse phase in the presence of high-contention workloads.

7 Concluding Remarks

We presented a novel read optimization for Paxos-style protocols with particular applications in distributed database settings. Our Paxos Quorum Reads (PQR) provides strong consistency guarantees without relying on external or orthogonal mechanisms. PQR performs reads against a set of follower nodes instead of running regular Paxos protocol or reading from dedicated leader. This allows PQR to balance the workload in the cluster and considerably improve performance over Paxos.

PQR can be extended with additional optimizations to provide a versatile solution for different workloads and scenarios. For example, we can use hash digest responses at all but one follower in the quorum-read phase to handle large objects better. Another optimization to consider is moving PQR inside the cluster to be run by the followers. This will make followers act as proxies/gateways to clients, and perform callbacks to the clients. This approach allows further optimizations, such as batching for read requests.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award number CNS-1527629 and XPS-1533870.

References

[1] A. Ailijiang. Paxos protocol framework. https://github.com/ailidani/paxi, 2018.

[2] A. Ailijiang, A. Charapko, and M. Demirbas. Dissecting the Performance of Strongly-Consistent Replication Protocols. *ACM SIGMOD/PODS Conference*, 2019.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 282–304. Springer-Verlag New York, Inc., 2003.

[4] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the shards: Managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460. USENIX Association, 2018.

[5] Apache ignite: Open source in-memory computing platform. https://zookeeper.apache.org/, 2017.

[6] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, and X. Xue. Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols. In *9th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2017.

[7] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, 2011.

[8] Citus Data. Master-less distributed queue with PG Paxos. https://www.citusdata.com/blog/2016/04/13/masterless-distributed-queue/, 2016.

[9] Cockroach Labs. Cockroachdb architecture overview. https://www.cockroachlabs.com/docs/stable/architecture/overview.html, 2018.

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

[11] Datastax Corporation. Introduction to Apache Cassandra. https://www.datastax.com/wp-content/uploads/2012/08/WP-IntrotoCassandra.pdf?1, August 2012.

[12] E. N. Hoch, Y. Ben-Yehuda, N. Lewis, and A. Vigder. Bizur: A key-value consensus algorithm for scalable file-systems. *arXiv preprint arXiv:1702.04242*, 2017.

[13] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[14] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[15] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[16] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 295–310. ACM, 2015.

[17] Microsoft. Cosmos DB - globally distributed, multi-model database service. https://azure.microsoft.com/en-us/services/cosmos-db/, 2018.

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

[19] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[20] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[22] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the annual conference on USENIX '09 Annual Technical Conference (ATC)*, pages 11–11, 2009.

[23] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42, 2015.

[24] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *6th*

Symposium on Operating Systems Design and Implementation (OSDI 04), pages 91–104. USENIX Association, 2004.

[25] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[26] J. Zheng, Q. Lin, J. Xu, C. Wei, C. Zeng, P. Yang, and Y. Zhang. PaxosStore: high-availability storage made practical in wechat. *Proceedings of the VLDB Endowment*, 10(12):1730–1741, 2017.