Read-Write Quorum Systems Made Practical

Michael Whittaker
mjwhittaker@berkeley.edu
UC Berkeley

Aleksey Charapko
aleksey.charapko@unh.edu
University of New Hampshire

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley

Heidi Howard
heidi.howard@cl.cam.ac.uk
University of Cambridge

Ion Stoica
istoica@berkeley.edu
UC Berkeley

Abstract

Quorum systems are a powerful mechanism for ensuring the consistency of replicated data. Production systems usually opt for majority quorums due to their simplicity and fault tolerance, but majority quorum systems provide poor throughput and scalability. Alternatively, researchers have invented a number of theoretically "optimal" quorum systems, but the underlying theory ignores many practical complexities such as machine heterogeneity and workload skew. In this paper, we conduct a pragmatic re-examination of quorum systems. We enrich the current theory on quorum systems with a number of practical refinements to find quorum systems that provide higher throughput, lower latency, and lower network load. We also develop a library Quoracle that precisely quantifies the available trade-offs between quorum systems to empower engineers to find optimal quorum systems, given a set of objectives for specific deployments and workloads.

1 Introduction

Ensuring the consistency of replicated data is a fundamental challenge in distributed computing. One widely utilized solution is to require that any operation over replicated data involve a quorum of machines. A **read-write quorum system** consists of a set of read quorums and a set of write quorums such that every read quorum and every write quorum intersect. Data is read from a read quorum of machines, and data is written to a write quorum of machines. This ensures that all previous writes are observed by subsequent reads. In addition to data replication, quorum systems have been applied to consensus algorithms [7, 10], distributed databases [14], abstract data types [6], mutual exclusion [2, 11] and shared memory [3] to name but a few.

Majority quorum systems—quorum systems where every read and write quorum consists of a strict majority of machines—are widely used in practice. Their simplicity makes them well-understood, and they also tolerate an optimal number of faults, $\left\lfloor \frac{n-1}{2} \right\rfloor$ with n machines. However, the performance of majority quorum systems is far from ideal. If each machine can process α commands per second then the maximum throughput of a majority quorum system is limited to just 2α, regardless of the number of machines [12].

The academic literature has responded by proposing many quorum systems including weighted voting [5], Finite Projective Planes [11], Trees [1], Hierarchies [9], Grids [4], B-Grids and Paths [12], and Crumbling Walls [13]. These sophisticated quorum systems are rarely used for two reasons. First, the theory behind these quorum systems ignores many practical considerations such as machine heterogeneity, workload skew, latency, and network load. As we show in Section 4, "theoretically optimal" quorum systems are often outperformed in practice. Second, understanding the various quorum systems and choosing the one that is optimal for a given workload is difficult and sensitive to workload parameters.

This paper is a practical re-examination of read-write quorum systems. We revisit the mathematical theory of quorum systems with a pragmatic lens and the ambition to make less-frequently used quorum systems more broadly accessible to the engineering community. More concretely, we make the following contributions:

1. We add a number of practical refinements to the theory of read-write quorum systems (§2). We extend definitions to accommodate heterogeneous machines and shifting workloads; we introduce the notion of f-resilient strategies to make it easier to trade off performance for fault tolerance; and we integrate metrics of latency and network load (§3).
2. We develop a Python library, called Quoracle (Quorum Oracle), that allows users to model, analyze, and optimize read-write quorum systems (§3). We also provide a heuristic search procedure to find quorum systems that are optimal given a number of objectives and constraints. Given the complex trade-off space, we believe that using an automated assistance library like ours is the only realistic way to find good quorum systems.
3. We perform a case study showing how to use Quoracle to find quorum systems that provide 2× higher throughput or 3× lower latency than naive majority quorums (§4).

2 Definitions

In this section, we present definitions adapted from existing theory on quorum systems [8, 12].

2.1 Read-Write Quorum Systems

Given a set $X = \{x_1, \ldots, x_n\}$, a **read-write quorum system** [12] over X is a pair $Q = (R, W)$ where

1. R is a set of subsets of X called **read quorums**,
2. W is a set of subsets of X called **write quorums**, and
3. every read quorum intersects every write quorum. That is, for every $r \in R$ and $w \in W$, $r \cap w \neq \emptyset$.

For example, the majority quorum system over the set $X = \{a, b, c\}$ is $Q_{\text{maj}} = (R, W)$ where $R = W = \{\{a, b\}, \{b, c\}, \{a, c\}\}$. If every read quorum intersects every write quorum, then any superset of a read quorum intersects any superset of a write quorum. Thus, if a set r is a superset of any read quorum in R, we consider r a read quorum as well. Similarly, if a set w is a superset of any write quorum in W, we consider w a write quorum. For example, we consider the set $\{a, b, c\}$ a read and write quorum of Q_{maj} even though the set $\{a, b, c\}$ is not listed explicitly in R or W.

It is notationally convenient to denote sets of read and write quorums over a set X as boolean expressions over X [8]. For example, we can represent the set $\{\{a, b\}, \{b, c\}, \{a, c\}\}$ as the expression $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$, which we abbreviate as $ab + bc + ac$. Equivalently, we can express the set as $a(b + c) + bc$, $b(a + c) + ac$, or $c(a + b) + ab$. As another example, consider the 2 by 3 grid quorum system $Q_{2\times3}$ over the set $X = \{a, b, c, d, e, f\}$ as shown in Figure 1. Every row is a read quorum, and every column is a write quorum. Concretely, $Q_{2\times3} = (abc + def, ad + be + cf)$.

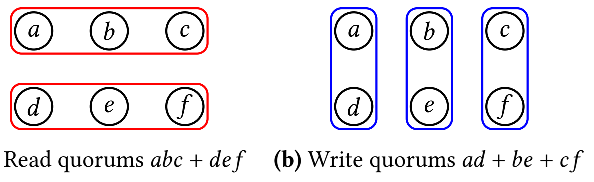

a
b
c
d
e
f
(a) Read quorums $abc + def$
(b) Write quorums $ad + be + cf$

Figure 1. The 2 by 3 grid quorum system $Q_{2\times3}$.

In practice, X might be a set of machines, a set of locks, a set of memory locations, and so on. In this paper, we assume that X is a set of machines we call **nodes**. We assume that protocols contact a read quorum of nodes to perform a read and contact a write quorum of nodes to perform a write.

2.2 Fault Tolerance

Unfortunately machines fail, and when they do, some quorums become unavailable. For example, if node a from the 2 by 3 grid quorum system $Q_{2\times3}$ fails, then the read quorum $\{a, b, c\}$ and the write quorum $\{a, d\}$ are unavailable. The **read fault tolerance** of a quorum system is the largest number f such that despite the failure of any f nodes, some read quorum is still available. **Write fault tolerance** is defined similarly, and the **fault tolerance** of a quorum system is the minimum of its read and write fault tolerance. For example, the read fault tolerance of $Q_{2\times3}$ is 1, the write fault tolerance is 2, so the fault tolerance is 1.

2.3 Load & Capacity

A protocol uses a **strategy** to decide which quorums to contact when executing reads and writes [12]. Formally, a strategy for a quorum system $Q = (R, W)$ is a pair $\sigma = (\sigma_R, \sigma_W)$ where $\sigma_R : R \to [0, 1]$ and $\sigma_W : W \to [0, 1]$ are discrete probability distributions over the quorums of R and W. $\sigma_R(r)$ is the probability of choosing read quorum r, and $\sigma_W(w)$ is the probability of choosing write quorum w. A **uniform strategy** is one where each quorum is equally likely to be chosen (i.e. $\sigma_R(r) = \frac{1}{|R|}$, $\sigma_W(w) = \frac{1}{|W|}$ for every r and w).

For a node $x \in X$, let $\text{load}_{\sigma_R}(x)$ be the probability that x is chosen by σ_R (i.e. the probability that σ_R chooses a read quorum that contains x). This is called the read load on x. Define $\text{load}_{\sigma_W}(x)$, the write load, similarly. Given a workload with a **read fraction** f_r of reads, the load on x is the probability that x is chosen by strategy σ and is equal to $f_r \text{load}_{\sigma_R}(x) + (1 - f_r)\text{load}_{\sigma_W}(x)$.

The most heavily loaded node is a throughput bottleneck, and its load is what we call the load of the strategy. The **load** of a quorum system is load of the optimal strategy (i.e. the strategy that achieves the lowest load). If a quorum system has load l, then the inverse of the load, $\frac{1}{l}$, is called the **capacity** of the quorum system. The capacity of a quorum system is directly proportional to the quorum system's maximum achievable throughput.

3 Practical Refinements

In this section, we augment the theory of read-write quorum systems with a number of practical considerations and demonstrate their use in our Python library Quoracle.

3.1 Quorum Systems, Capacity, Fault Tolerance

Quoracle allows users to form arbitrary read-write quorum systems and compute their capacity and fault tolerance. For example, in Figure 2, we construct and analyze the majority quorum system on nodes $\{a, b, c\}$. As in Section 2.1, read-write quorum systems are constructed from a set of read or write quorums expressed as a boolean expression over the set of nodes. The user only has to specify one set of quorums rather than both because we automatically construct the optimal set of complementary quorums [8]. We compute fault tolerance and capacity using linear programming [12].

```
a, b, c = Node('a'), Node('b'), Node('c')
majority = QuorumSystem(reads=a*b + b*c + a*c)
print(majority.fault_tolerance())          # 1
print(majority.load(read_fraction=1))      # 2/3
print(majority.capacity(read_fraction=1)) # 3/2
```

Figure 2. Quorum systems, capacity, and fault tolerance.

3.2 Heterogeneous Nodes

Quorum system theory implicitly assumes that all nodes are equal. In reality, nodes are often heterogeneous. Some are fast, and some are slow. Moreover, nodes can often process more reads per second than writes per second. We revise the theory by associating every node x with its read and write capacity, i.e. the maximum number of reads and writes the node can process per second. We redefine the read load imposed by a strategy $\sigma = (\sigma_R, \sigma_W)$ on a node x as the probability that σ_R chooses x divided by the read capacity of x. We redefine the write load similarly. By normalizing a node's load with its capacity, we get a more intuitive definition of a quorum system's capacity. Now, the capacity of a quorum system is the maximum throughput that it can support.

Quoracle allows users to annotate nodes with read and write capacities. For example, in Figure 3, we construct a 2 by 2 grid quorum system where nodes a and b can process 100 writes per second, but nodes c and d can only process 50 writes per second. We also specify that every node can process reads twice as fast as writes. With a read fraction of 1, the quorum system has a capacity of 300 commands per second using a strategy that picks the read quorum $\{a, b\}$ twice as often as the read quorum $\{c, d\}$. As we decrease the fraction of reads, the capacity decreases since the nodes process reads faster than writes.

```
a = Node('a', write_cap=100, read_cap=200)
b = Node('b', write_cap=100, read_cap=200)
c = Node('c', write_cap=50, read_cap=100)
d = Node('d', write_cap=50, read_cap=100)
grid = QuorumSystem(reads=a*b + c*d)
print(grid.capacity(read_fraction=1))   # 300
print(grid.capacity(read_fraction=0.5)) # 200
print(grid.capacity(read_fraction=0))   # 100
```

Figure 3. Heterogeneous nodes with different capacities.

3.3 Workload Distributions

Capacity is defined with respect to a fixed read and write fraction, but in reality, workloads skew. To accommodate workload skew, we consider a discrete probability distribution over a set of read fractions and redefine the capacity of a quorum system to be the capacity of the strategy σ that maximizes the expected capacity with respect to the distribution. For example, in Figure 4, we construct the quorum system with read quorums $ac + bd$, and we define a workload that has 0% reads $\frac{10}{18}$th of the time, 25% reads $\frac{4}{18}$th of the time, and so on. In Figure 4, we see the optimal strategy σ has an expected capacity of 159 commands per second.

In Figure 5, we plot σ's capacity as a function of read fraction. We also plot the capacities of strategies $\sigma_{0.0}$, $\sigma_{0.25}$, $\sigma_{0.50}$, $\sigma_{0.75}$, and $\sigma_{1.0}$ where σ_{f_r} is the strategy optimized for a fixed workload with a read fraction of f_r. We see that

```
grid = QuorumSystem(reads=a*c + b*d)
fr = {0.00: 10/18, 0.25: 4/18, 0.50: 2/18,
      0.75:  1/18, 1.00: 1/18}
sigma = grid.strategy(read_fraction=fr)
print(sigma.capacity(read_fraction=fr)) # 159
```

Figure 4. A distribution of read fractions.

σ does not always achieve the maximum capacity for any *individual* read fraction, but it achieves the best expected capacity across the distribution.

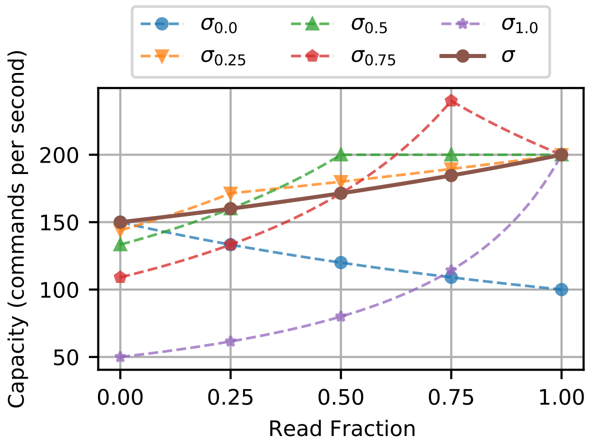


Figure 5. Strategy capacities with respect to read fraction

Note that σ performs well across all workloads drawn from the distribution without having to know the current read fraction. Alternatively, if we are able to monitor the workload and deduce the current read fraction, we can precompute a set of strategies that are optimized for various read fractions and dynamically select the one that is best for the current workload.

3.4 f-resilient Strategies

Many protocols that deploy read-write quorum systems actually contact more nodes than is strictly necessary when executing a read or a write. Rather than contacting a quorum to perform a read or write, these protocols contact *every* node. Contacting every node leads to suboptimal capacity, but it is less sensitive to stragglers and node failures. For example, if we contact only a quorum of nodes and one of the nodes in the quorum fails, then we have to detect the failure and contact a different quorum. This can be slow and costly. Typically, industry practitioners have chosen between these two extremes: either send messages to *every node* or send messages to the *bare minimum number of nodes* (i.e. a quorum). We introduce the notion of f-resilient strategies to show that this is not a binary decision, but rather a continuous trade-off.

Given a quorum system (R, W), we say a read quorum $r \in R$ is f-resilient for some integer f if despite remove any f nodes from r, r is still a read quorum. We define f-resilience for write quorums similarly. We say a strategy σ is **f-resilient** if it only selects f-resilient read and write quorums. An f-resilient strategy can tolerate any f failures or stragglers. The value of f captures the continuous trade-off between capacity and resilience. As we increase f, we decrease capacity but increase resilience.

Quoracle allows users to compute optimal f-resilient strategies and their corresponding capacities. For example, in Figure 6, we compute the f-resilient capacity of a grid quorum system for $f = 0$ and $f = 1$. Its 0-resilient capacity is 300, but its 1-resilient capacity is only 100. We then compute the f-resilient capacities for the "read 2, write 3" quorum system. For this quorum system, every set of two nodes is a read quorum, and every set of three nodes is a write quorum. This quorum system has the same 0-resilient capacity as the grid but a higher 1-resilient capacity showing that some quorum systems are naturally more resilient than others.

```
grid = QuorumSystem(reads=a*b + c*d)
print(grid.capacity(read_fraction=1, f=0)) # 300
print(grid.capacity(read_fraction=1, f=1)) # 100
read2 = QuorumSystem(reads=choose(2, [a, b, c, d]))
print(read2.capacity(read_fraction=1, f=0)) # 300
print(read2.capacity(read_fraction=1, f=1)) # 200
```

Figure 6. 0-resilient and 1-resilient strategies.

3.5 Latency and Network Load

Quorum system theory focuses on capacity and fault tolerance, but these are not the only metrics that are important in practice. We introduce two new practically important metrics. First, we introduce **latency**. We associate every node with a latency that represents the time required to contact the node. The latency of a quorum q is the time required to form a quorum of responses after contacting the nodes in q. The latency of a strategy is the expected latency of the quorums that it selects. The lower the latency, the better.

Second, we introduce **network load**. When a protocol executes a read or write, it sends messages over the network to every node in a quorum, so as the sizes of quorums increase, the number of network messages increases. The network load of a strategy is the expected size of the quorums it chooses. The lower the network load, the better.

In isolation, optimizing for latency or network load is trivial, but balancing capacity, fault tolerance, latency, and network load simultaneously is very complex. Quoracle allows users to find strategies that are optimal with respect to load, latency, or network load with constraints on the other metrics. For example, in Figure 7, we specify the latencies of the nodes in our 2 by 2 grid quorum system and then find the latency optimal strategy with a capacity no less than 150 and with a network load of at most 2.

```
a = Node('a', write_cap=100, read_cap=200, latency=4)
b = Node('b', write_cap=100, read_cap=200, latency=4)
c = Node('c', write_cap=50, read_cap=100, latency=1)
d = Node('d', write_cap=50, read_cap=100, latency=1)
grid = QuorumSystem(reads=a*b + c*d)
grid.strategy(read_fraction = 1,
              optimize = 'latency',
              capacity_limit = 150,
              network_limit = 2)
```

Figure 7. Finding a latency-optimal strategy with capacity and network load constraints.

In reality, the relationships between these metrics are more complex. As the load on a node increases, for example, the latencies of the requests sent to it increase. We leave these complexities to future work.

3.6 Quorum System Search

Thus far, we have demonstrated how Quoracle makes it easy to model, analyze, and optimize a *specific hand-chosen* quorum system. Quoracle also implements a heuristic based search procedure to find good quorum systems. For example, in Figure 8, we search for the latency-optimal quorum system over the nodes $\{a, b, c, d\}$ with a capacity of at least 150 and a network load of at most 2.

```
search(nodes = [a, b, c, d],
       read_fraction = 1,
       optimize = 'latency',
       capacity_limit = 150,
       network_limit = 2)
```

Figure 8. Searching the space of quorum systems.

4 Case Study

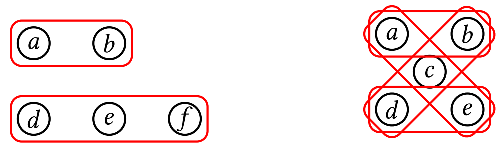
In this section, we present a case study that demonstrates how to use Quoracle in a realistic setting. We have five nodes. Nodes a, c, and e can process 2,000 writes per second, while nodes b and d can only process 1,000 writes per second. All five nodes process reads twice as fast as writes. Nodes a and b have a latency of 1 second, while nodes c, d, and e have latencies of 3, 4, and 5 seconds. We observe a workload with roughly equal amounts of reads and writes with a slight skew towards being read heavy. In Figure 9, we use Quoracle to model the nodes and workload distribution.

Assume we have already deployed a majority quorum system with a uniform strategy, which has a capacity of 2,292 commands per second. We want to find a more load optimal quorum system. We consider three candidates. The first is

```
a = Node('a', write_cap=2000, read_cap=4000, latency=1)
b = Node('b', write_cap=1000, read_cap=2000, latency=1)
c = Node('c', write_cap=2000, read_cap=4000, latency=3)
d = Node('d', write_cap=1000, read_cap=2000, latency=4)
e = Node('e', write_cap=2000, read_cap=4000, latency=5)
fr = {0.9:  10/470, 0.8:  20/470, 0.7: 100/470,
      0.6: 100/470, 0.5: 100/470, 0.4:  60/470,
      0.3:  30/470, 0.2:  30/470, 0.1:  20/470}
```

Figure 9. Nodes and workload distribution.

the majority quorum system. The second is a staggered grid quorum system, illustrated in Figure 10a. The third is a quorum system based on paths through a two-dimensional grid illustrated in Figure 10b. This quorum system has theoretically optimal capacity [12]. In Figure 11, we construct these three quorum systems and print their capacities.



(a) Staggered grid quorum system **(b)** Paths quorum system

Figure 10. The read quorums of the staggered grid and paths quorum systems. The optimal set of complementary write quorums is chosen automatically.

```
maj = QuorumSystem(reads=majority([a, b, c, d, e]))
grid = QuorumSystem(reads=a*b + c*d*e)
paths = QuorumSystem(reads=a*b + a*c*e + d*e + d*c*b)
print(maj.capacity(reads_fraction=fr))    # 3,667
print(grid.capacity(reads_fraction=fr))  # 4,200
print(paths.capacity(reads_fraction=fr)) # 4,125
```

Figure 11. Quorum systems and their capacities.

The capacities are 3,667, 4,200, and 4,125 commands per second respectively, making the grid quorum system the most attractive. However, the grid quorum system is not necessarily optimal. In Figure 12, we perform a search for a load optimal quorum system that is tolerant to one failure. The search takes 7 seconds on a laptop.

```
qs = search(nodes=[a, b, c, d, e],
            fault_tolerance=1,
            read_fraction=fr)
print(qs.capacity(read_fraction=fr)) # 5,005
```

Figure 12. Searching for a load-optimal quorum system.

The search procedure finds the quorum system with read quorums $(c + bd)(a + e)$ which has a capacity of 5,005 commands per second. This is 1.19× better than the grid quorum system, and 2.18× better than the majority quorum system with a naive uniform strategy. We deploy this strategy to production. Months later, we introduce a different component into our system that bottlenecks our throughput at 2,000 commands per second. Now, any capacity over 2,000 is wasted, so we search for a latency optimal quorum system that has a capacity of at least 2,000. We again consider our three candidate quorum systems in Figure 13.

```
for qs in [maj, grid, paths]:
    print(qs.latency(read_fraction=fr,
                     optimize='latency',
                     capacity_limit=2000))
```

Figure 13. Latencies with a capacity constraint.

The quorum systems have latencies of 3.24, 1.95, and 2.43 seconds respectively, making the grid quorum system the most attractive. We again perform a search and find the quorum system with read quorums $ab+acde+bcde$ achieves a latency of 1.48 seconds. This is 1.32× better than the grid and 3.04× better than a naive uniform strategy over a majority quorum system. The search again completes in 7 seconds. We deploy this quorum system to production.

Lesson 1: Naive Majority Quorums Underperform. Industry practitioners often use majority quorums because they are simple and have strong fault tolerance. Our case study shows that majority quorum systems with uniform strategies almost always underperform more sophisticated quorum systems in terms of capacity, latency, and network load. In Figure 14, we plot the throughput that every node in a majority quorum system obtains using a naive uniform strategy, with throughput broken down by quorums. We contrast this with the strategy found by our heuristic search procedure, illustrated in Figure 15. The sophisticated quorum system assigns more work to machines with higher capacities, leading to a 2.18× increase in aggregate throughput.

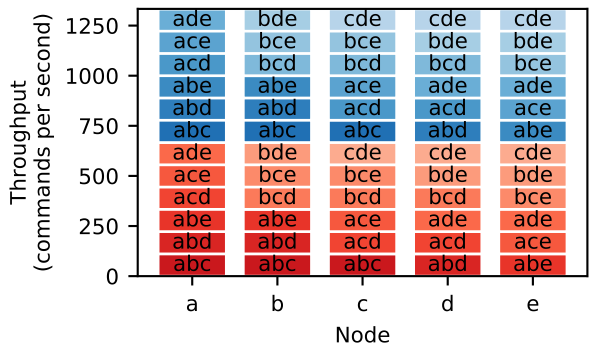


Figure 14. The throughput of a simple majority quorum system with a naive uniform strategy. Write quorums are in blue, and read quorums are in red.

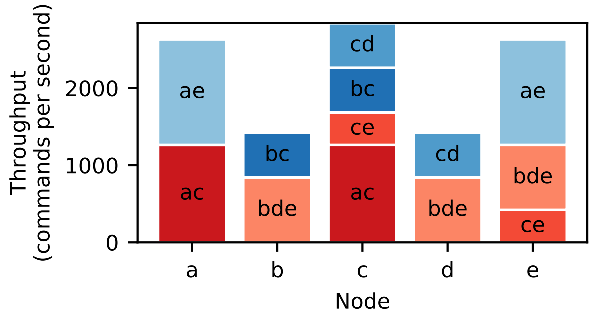


Figure 15. The throughput of the quorum system found by our heuristic search (i.e., the quorum system with read quorums $(c + bd)(a + e)$).

Lesson 2: Optimal Is Not Always Optimal. There is a large body of research on constructing "optimal" quorum systems [1, 4, 5, 9, 11–13]. For example, the paths quorum system is theoretically optimal, but in our case study, it has lower capacity and higher latency than the simpler grid quorum system. There are two reasons for this mismatch between theoretically and practically optimal. First, existing quorum system theory does not account for node heterogeneity and workload skew. Second, these quorum systems are only optimal in the limit, as the number of nodes tends to infinity.

Lesson 3: The Trade-Off Space Is Complex. Constructing a quorum system of homogeneous nodes that is optimal in the limit for a fixed workload is difficult but doable. When nodes operate at different speeds and workloads skew, finding an optimal quorum system that satisfies constraints on capacity, fault tolerance, latency, and network load becomes nearly impossible to do by hand. Moreover, small perturbations in any of these parameters can change the landscape of the optimal quorum systems. In our case study, for example, the search procedure finds two different quorum systems when optimizing for load and when optimizing for latency. We believe that using an automated assistance library like ours is the only realistic way to find good quorum systems.

5 Conclusion

Majority quorum systems have garnered the most attention due to their simplicity and well-understood properties. Our tool, called Quoracle, allows engineers to find more optimal quorum systems and strategies for a given set of constraints. With our tool, we not only show that majority quorum systems are not necessarily the best, but we also illustrate that some quorums that theoretically boast better performance are likely to underperform in practical conditions.

While the tool is easy to use, it has a few limitations, which we will address in the future. A more sophisticated latency model would help improve the computation of latency-optimized strategies. We could also incorporate other practical features such as a cost calculator and the ability to run the tool as a service to let applications adopt quorum systems on the fly. Our tool and the associated scripts needed to reproduce this paper's calculations are available at: https://github.com/mwhittaker/quoracle.

References

[1] Divyakant Agrawal and Amr El Abbadi. 1990. The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 243–254.

[2] D. Agrawal and A. El Abbadi. 1989. Efficient Solution to the Distributed Mutual Exclusion Problem. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alberta, Canada) *(PODC '89)*. Association for Computing Machinery, New York, NY, USA, 193–200. https://doi.org/10.1145/72981.72994

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (Jan. 1995), 124–142. https://doi.org/10.1145/200836.200869

[4] S. Y. Cheung, M. H. Ammar, and M. Ahamad. 1992. The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. *IEEE Trans. on Knowl. and Data Eng.* 4, 6 (Dec. 1992), 582–592. https://doi.org/10.1109/69.180609

[5] Hector Garcia-Molina and Daniel Barbara. 1985. How to Assign Votes in a Distributed System. *J. ACM* 32, 4 (Oct. 1985), 841–860. https://doi.org/10.1145/4221.4223

[6] Maurice Herlihy. 1986. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Trans. Comput. Syst.* 4, 1 (Feb. 1986), 32–53. https://doi.org/10.1145/6306.6308

[7] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[8] Toshihide Ibaraki and Tiko Kameda. 1993. A theory of coteries: Mutual exclusion in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 4, 7 (1993), 779–794.

[9] Akhil Kumar. 1991. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. *IEEE Trans. Comput.* 40, 9 (Sept. 1991), 996–1004. https://doi.org/10.1109/12.83661

[10] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[11] Mamoru Maekawa. 1985. A square root N Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Trans. Comput. Syst.* 3, 2 (May 1985), 145–159. https://doi.org/10.1145/214438.214445

[12] Moni Naor and Avishai Wool. 1998. The Load, Capacity, and Availability of Quorum Systems. *SIAM J. Comput.* 27, 2 (1998). https://doi.org/10.1137/S0097539795281232

[13] David Peleg and Avishai Wool. 1995. Crumbling Walls: A Class of Practical and Efficient Quorum Systems. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 120–129. https://doi.org/10.1145/224964.224978

[14] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.* 4, 2 (June 1979), 180–209. https://doi.org/10.1145/320071.320076