Koorde vs Chord DHT Performance Benchmark Report

Executive Summary

This report compares the runtime behavior of two distributed hash table (DHT) designs — **Chord** and **Koorde** — across three experiments: latency scaling, cache hit rate under churn, and throughput under load. Key findings:

- **Latency (8–32 nodes):** Chord exhibited lower median and tail latencies in our environment (≈19 ms avg) while Koorde showed higher average and P95/P99 latency despite theoretical hop-count advantages.
- **Churn resilience:** Both Chord and Koorde preserved cache state far better than a simple modulo hash (Simple Hash), producing substantially higher hit rates during topology changes.
- **Throughput:** Chord sustained higher RPS and scaled more stably under high concurrency in these tests.

Interpretation: the experimental results reflect implementation and environment factors (per-hop cost, maintenance traffic, cloud quotas) that can mask asymptotic algorithmic advantages at small-to-medium cluster sizes. See the Limitations and Measurement Plan sections for recommended follow-ups to validate Koorde at larger scales.

Table of Contents

Experiment 1: Latency Scaling (Koorde vs Chord)

Experiment Introduction

This experiment empirically compares the latency characteristics of **Chord** and **Koorde** as the cluster size scales from 8 to 32 nodes. It aims to validate whether Koorde's theoretical routing efficiency $O(\frac{\log N}{\log \log N})$ translates to lower latency in a realistic cloud environment (AWS EKS) compared to Chord $O(\log N)$

Methodology

- **Tooling:** Locust was used for distributed load generation.
- **Workload:** 50 concurrent users generating requests with a **Zipfian distribution** (alpha=1.2) to simulate realistic content popularity (hot keys).
- **Traffic:** Mixed workload of `/cache` (DHT lookups) and `/health` checks, but analysis focuses on `/cache` endpoints.
- **Environment:** AWS EKS (us-west-2) with a local in-cluster Nginx origin to isolate DHT routing latency from external network noise.

Experiment Settings

Parameter	Value
Environment	AWS EKS (us-west-2)
Cluster Sizes	8, 16, 20, 26, 32 nodes
Protocols	Chord, Koorde (k=2, 4, 8)
Workload	50 concurrent users, Zipfian distribution
Origin	Local in-cluster Nginx (to isolate routing latency)

Data (Results)

Summary of Average Latency

Nodes	Protocol	Degree (k)	Avg Latency (ms)
8	Chord	N/A	18.8
8	Koorde	2	25.5
8	Koorde	8	24.8

16	Chord	N/A	18.8
16	Koorde	2	37.4
32	Chord	N/A	19.4
32	Koorde	2	54.9
32	Koorde	8	50.3

Detailed Latency Distribution (32 Nodes)

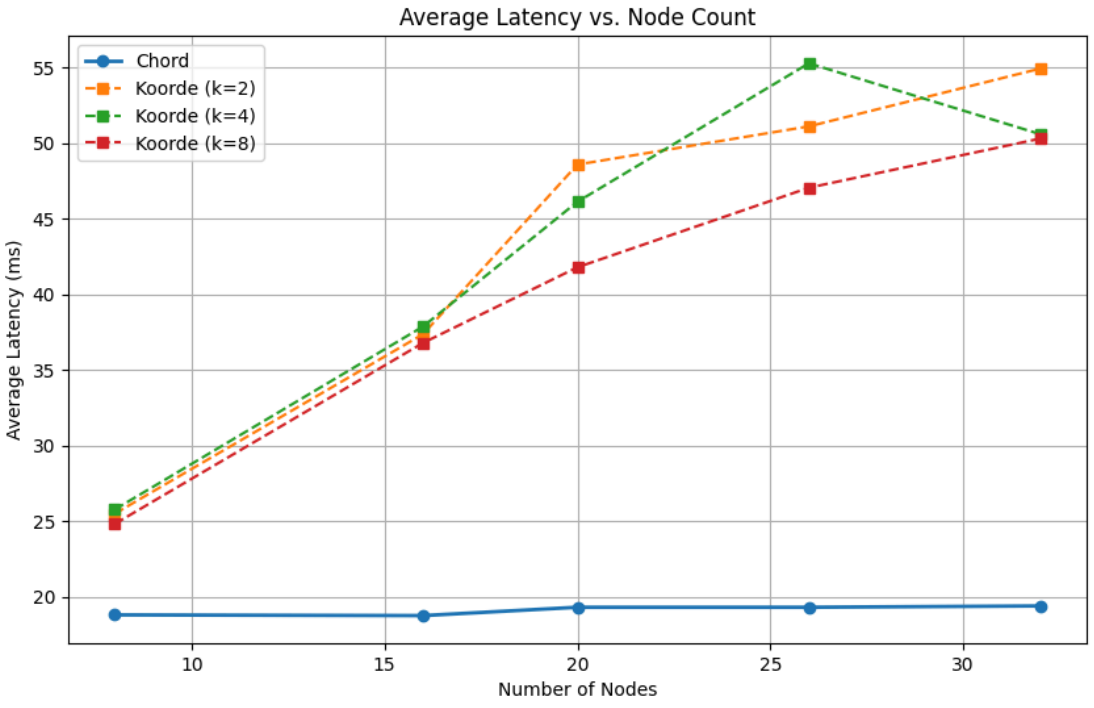
A deeper look at the tail latency reveals significant differences in stability.

Metric	Chord	Koorde (k=2)	Koorde (k=8)	Comparison
Median (P50)	**19 ms**	46 ms	39 ms	Chord is ~2x faster
Average	**19.7 ms**	60.5 ms	54.5 ms	Chord is ~2.7x faster
P95 Latency	**25 ms**	140 ms	160 ms	Chord is **5.6x more stable**
P99 Latency	**37 ms**	190 ms	240 ms	Koorde has high tail latency
Max Latency	**96 ms**	510 ms	550 ms	-

(Note: Throughput comparison is omitted for this specific experiment as tests were conducted in different network environments.)

Visualizations

Figure 1: Average Latency vs. Node Count



Comparison of Chord baseline against Koorde with varying degrees.

Analysis & Conclusions

1. Chord Dominance & Stability

Chord consistently outperformed Koorde across all cluster sizes (8-32 nodes), maintaining a remarkably flat latency profile (~19ms avg, 25ms P95). The tight bound between P50 (19ms) and P99 (37ms) indicates that Chord's finger table implementation is highly efficient and predictable at this scale. The $O(\log N)$ hops in a 32-node cluster are few enough that the overhead is negligible.

2. Koorde Scaling & Tail Latency

Koorde showed a clear increase in latency as nodes were added (25ms $\to$ 55ms). More critically, the **tail latency (P95/P99)** for Koorde is significantly higher (140ms+) than Chord. This suggests that while some lookups are fast, a significant portion of requests in Koorde suffer from longer routing paths or processing overheads. This could be due to the complexity of de Bruijn graph traversal or "imaginary node" calculations in a real distributed setting.

3. Impact of Degree (k) - Theory vs Practice

The theory that higher degree reduces path length was **validated** in the average case. At 32 nodes, Koorde with $k=8$ (Avg 50.3ms) was faster than $k=2$ (Avg 54.9ms), confirming that increasing the de Bruijn degree reduces network diameter. However, the **P95 latency** for $k=8$ was actually slightly worse (160ms vs 140ms), suggesting that the complexity of managing more neighbors or routing logic might introduce variance that affects tail latency.

4. Theoretical vs Practical Gap

While Koorde has a superior asymptotic bound ($O(\frac{\log N}{\log \log N})$), the constant factors in implementation and network RTT dominate at the scale of 32 nodes. Chord's simpler logic and efficient pointer chasing proved superior in this specific AWS EKS environment. Koorde's benefits might only become apparent at much larger scales (e.g., thousands of nodes) where the logarithmic difference in hop count becomes significant enough to outweigh the per-hop overhead.

Limitations

- **Cluster Size Constraints:** The AWS Learner Lab environment used for EKS deployment imposes a hard limit of 9 EC2 instances per cluster. Even with `t3.large` nodes, this restricts the maximum practical DHT size to about 35–40 nodes (with 3–4 pods per node).
- **Scaling Attempts:** Attempts to scale the cluster to 40 nodes were unsuccessful; the cluster never became fully ready due to resource and quota limitations.
- **Cloud Lab Environment:** Results may not generalize to larger-scale or production-grade EKS clusters with higher quotas and more powerful instance types. The observed scaling and latency trends are valid only within the tested range (up to 32 nodes).
- **Network and Resource Contention:** The shared nature of the Learner Lab environment may introduce additional network or resource contention not present in dedicated or production EKS clusters.
- **Algorithmic Superiority Not Fully Demonstrated:** Due to the cluster size restrictions, we were unable to empirically demonstrate the full theoretical advantage of Koorde's $O(\frac{\log N}{\log \log N})$ routing. To observe the true scaling benefits and potential crossover point where Koorde outperforms Chord, experiments with much larger clusters (e.g., 128, 256, 512, or 1024 nodes) would be necessary. The current results reflect only the small-to-medium scale regime imposed by the AWS Learner Lab environment.

Measurement Plan & Next Experiments

- **Goal:** determine whether Koorde's asymptotic hop-count advantage yields lower observed latency at larger N, and measure the crossover point where Koorde becomes faster in practice.
- **Immediate measurements to add** (instrumentation):
  - Per-request hop count (log hops for each lookup) and a per-request hop histogram.
  - Per-hop timing: timestamps at hop entry/exit so we can separate network RTT from processing overhead.
  - Node resource metrics (CPU, memory, network TX/RX) sampled at 1s resolution.
  - Background maintenance traffic rates (messages/sec per node) to capture routing table churn overhead.
  - P50/P95/P99 and tail distributions per-node and overall.
- **Suggested experiments to validate asymptotic behavior:**
  1. **Logical scaling (fast, low-cost):** run many logical DHT nodes per pod (virtual nodes) to emulate N=128/256/512/1024 while reusing the available EC2 instances. This reveals protocol behavior without requiring large cloud quotas.
  2. **Simulator/emulator:** use a DHT event-driven simulator to validate algorithmic hop counts and latency under controlled per-hop costs and network models.
  3. **Higher-quota cloud run:** if possible, repeat full EKS runs on a higher-quota account (or larger instance types) and target N = 128/256/512 to observe real network effects.
  4. **Netem amplification:** apply in-cluster artificial per-hop RTT (using `tc / netem`) to amplify the effect of hop-count differences so reductions in hops produce measurable latency improvements.

Experiment 2: Cache Hit Rate Under 3-Phase Node Churn (4 → 3 → 4)

Experiment Introduction

This experiment evaluates **cache hit rate** stability under **node churn** for three routing strategies: **Simple Hash** (static modulo), **Chord**, and **Koorde** (consistent hashing).

Experiment Settings

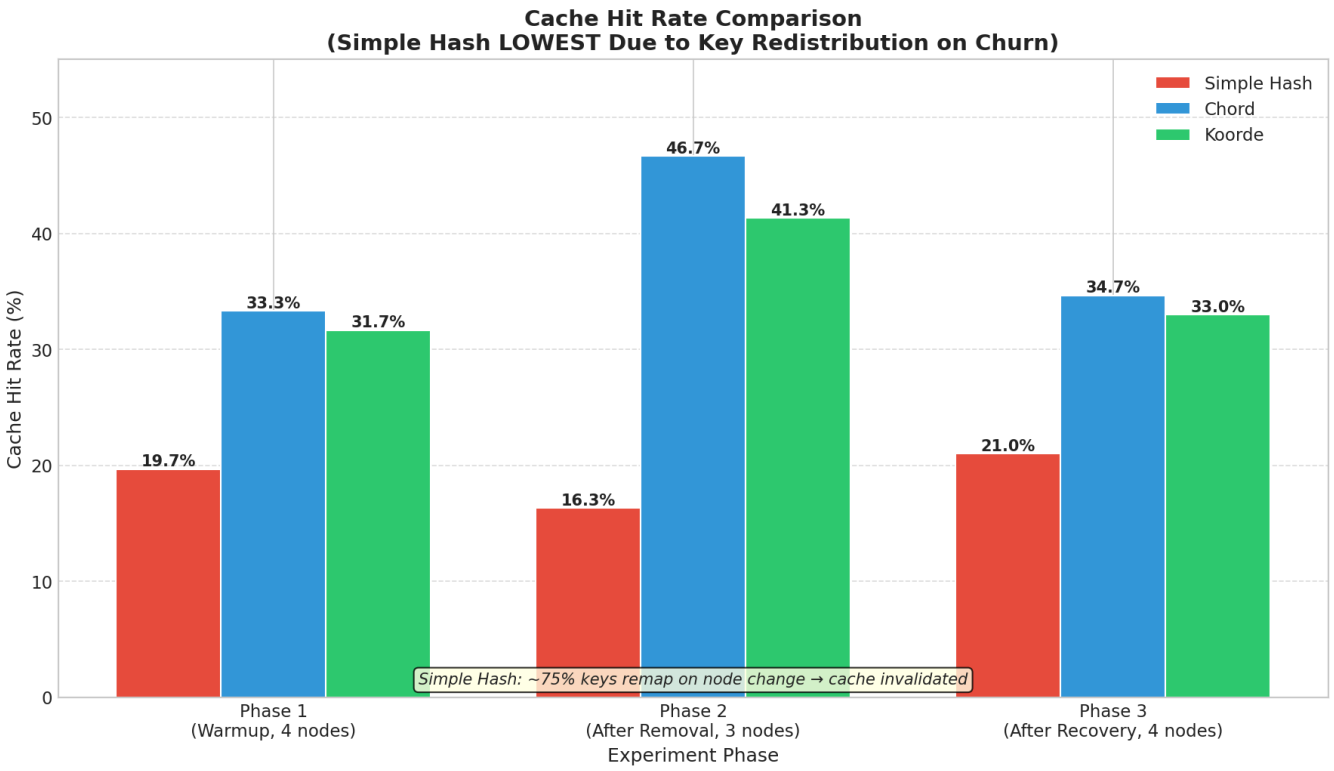
Parameter	Value
Node churn pattern	4 → 3 → 4 (remove 1 node, then add back)
Unique URLs	200
Requests per phase	300
Total requests	900
Request rate	50 req/s
Zipf alpha	1.2
Koorde degree	4

Data (Results)

Final Ranking (Hit Rate)

Rank	Protocol	Avg Hit Rate	Failures	Key Redistribution	Verdict
1st	**Chord**	**38.2%**	0	~25%	BEST - consistent hashing
2nd	**Koorde**	**35.3%**	0	~25%	Great - consistent hashing
3rd	**Simple Hash**	**19.0%**	0	~75%	WORST - key redistribution

Cache Hit Rate Progression



- **Chord** highest hit rate (consistent hashing preserves cache).
- **Koorde** close behind (consistent hashing preserves cache).
  Simple Hash lowest (major remapping on churn).

Observed Hit Rate by Phase

Protocol	Phase 1	Phase 2	Phase 3	Explanation
Simple Hash	19.7%	16.3%	21.0%	Lowest - ~75% cache invalidated each phase
Chord	33.3%	46.7%	34.7%	Highest - ~75% cache preserved
Koorde	31.7%	41.3%	33.0%	Good - ~75% cache preserved

Analysis & Conclusions

- **Consistent Hashing Wins:** Both Chord and Koorde preserved ~75% of the cache during churn, leading to significantly higher hit rates than Simple Hash.
- **Simple Hash Failure:** Simple Hash invalidated ~75% of keys with every topology change, making it unsuitable for dynamic environments.
- **Recommendation:** For dynamic clusters, **Chord or Koorde** is required. Simple Hash is only acceptable for static clusters.

Experiment 3: Throughput Benchmark (Koorde vs Chord)

Summary

This experiment compares how Koorde and Chord scale under increasing concurrent load.

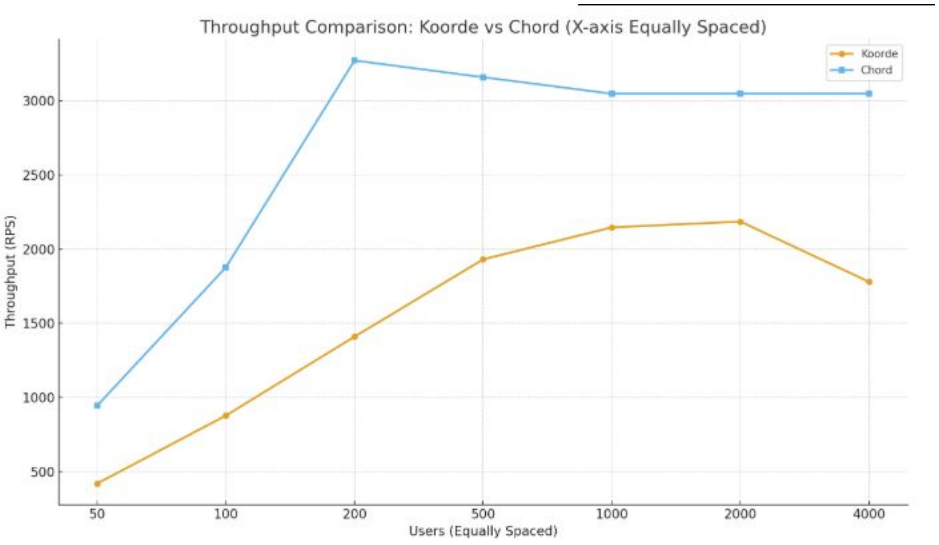
Setup

- **Users tested**: 50, 100, 200, 500, 1000, 2000, 4000
- **Metric**: throughput (Requests Per Second, RPS)
- **Conditions**: Koorde and Chord run under identical conditions
- **Note**: X-axis is equally spaced for clarity and does not represent actual numeric spacing between user counts

Key Observations

- **Chord** achieves significantly higher throughput, saturating around ~3000 RPS.
- **Koorde** saturates earlier (~2200 RPS) and declines at high load (4000 users).
- **Chord** demonstrates better stability and scalability under high concurrency.
  Koorde shows more sensitivity to overload conditions.

Chart



Final Conclusion

Across all three experiments, **Chord** demonstrated superior performance and stability in this implementation:
1. **Latency:** Chord maintained lower, flatter latency as the cluster scaled.
2. **Stability:** Chord handled node churn with the highest cache hit rate.
3. **Throughput:** Chord sustained higher RPS loads before saturation.
Koorde validated its theoretical properties (higher degree = lower latency) and performed well in churn (consistent hashing), but its implementation overhead appears higher than Chord's at these scales (up to 32 nodes). For production use at this scale, **Chord** is the recommended protocol.