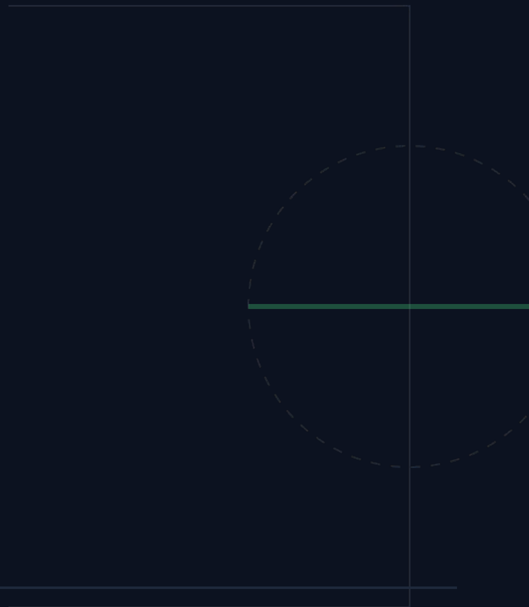
CONF42 / DATABASE DEVOPS 2026

Deterministic Routing for Data Locality

in Distributed Databases

> *How to achieve 2x throughput and 59% P99 reduction in a transactional platform*



Ritvik Pandya | Engineering Manager

A familiar problem

Peak traffic, random routing, latency spikes

THE PROBLEM

Service API P99 spikes from 12ms to 47ms during peak.

error_outline Transactions timing out.

storm Retry storms amplifying load.

notification_important SLA breaches triggering alerts.

THE ROOT CAUSE

Random node selection.

- Every transaction picks a random node.
- Data isn't where the request lands.
- Cross-node coordination on every write.
- Cache misses, lock contention, and latency.

"What if every transaction landed on the node that already has its data?"

Why every millisecond matters

In transactional, latency isn't just a metric — it's revenue

47ms

P99 during peak

Baseline was 12ms. 4x increase under load.

3.2%

Request timeouts

Each timeout is a failed request and a retry.

$0.12

Per failed request

Retry costs, customer friction, SLA penalties.

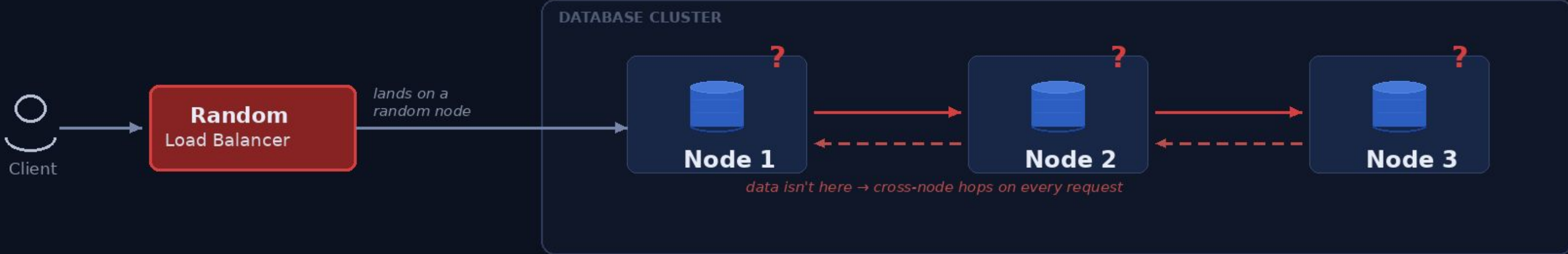
2x

Throughput needed

Business growing. Same infra. Need to scale without adding nodes.

The baseline architecture

Random routing, single-threaded, default Kubernetes



Redundant metadata lookups

Every request resolves data location from scratch

Cross-node replication on every write

Synchronous coordination inflates commit times

Underutilized compute

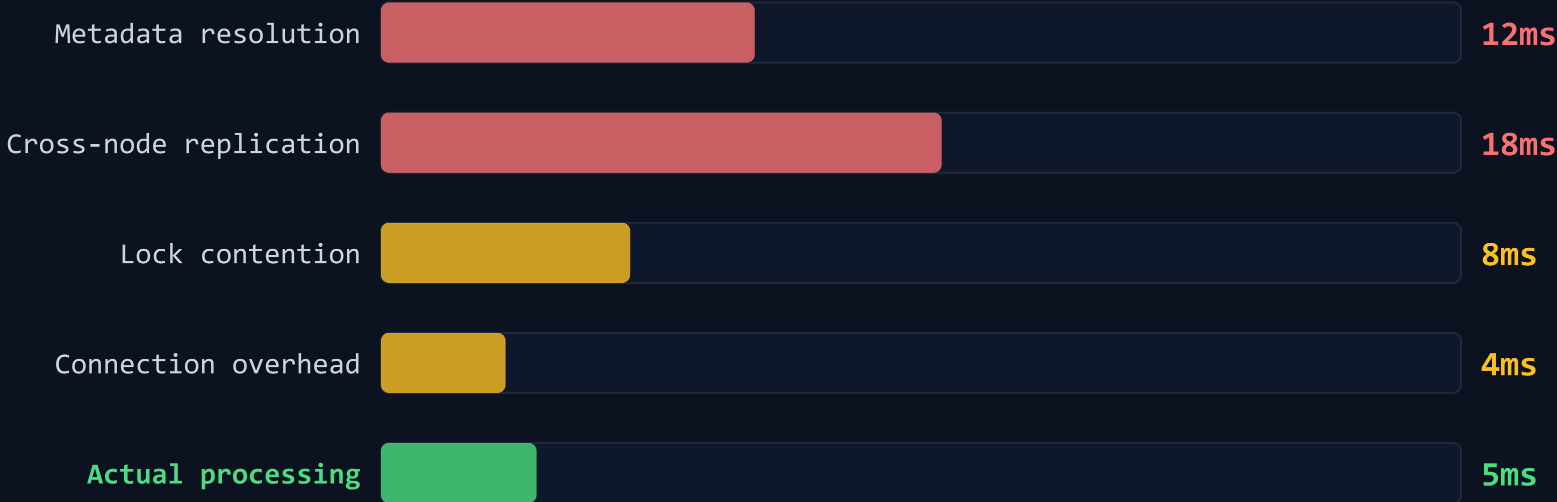
Some nodes bottleneck, others sit idle

Contention under concurrency

Lock conflicts and race conditions at peak load

Where the time goes

Latency breakdown of the baseline architecture (sums to 47ms P99)



Only ~10% of P99 latency is actual processing. The rest is coordination overhead (metadata, replication, contention, queueing) that deterministic routing eliminates.

The Thesis

One architectural change. Everything else follows from it.

Route every transaction to the node that

already has its data,

using deterministic key hashing,
with parallelized execution.

Reduce Coordination Overhead

Significantly minimizes the need for cross-node communication and locking.

Data Locality Primitive

Treats proximity of computation to data as a first-class design requirement.

Key takeaway: Efficiency is gained by eliminating unnecessary motion of data. (Deterministic scheduling: Calvin, SIGMOD 2012.)

Deterministic node mapping via key hashing

Predictable routing, no request-path metadata lookups

THE FORMULA

```
node_index = hash(primary_key) % total_nodes
selected_node = node_list[node_index]
```

Predictable routing

Same key routes to same node group. Deterministic.

No runtime lookups

Target node computed from the key itself.

Max cache locality

Repeated access = hot cache, minimal disk I/O.

Even distribution

Spreads load uniformly across nodes.

The key drives the routing decision. Production uses consistent hashing ring with virtual nodes — not simple modulo. Minimizes remapping during cluster changes. *(Karger et al. 1997; Dynamo 2007.)*

Parallelized, node-aware execution

Multi-threaded workers with thread-to-node affinity

Thread-to-node affinity

Each worker thread owns a connection pool to its assigned node group. No cross-thread contention.

Round-robin distribution

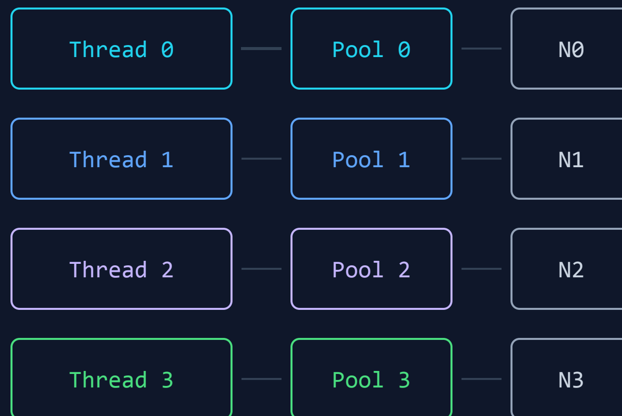
Incoming transactions spread evenly across threads. No starvation, no overload.

Async I/O per thread

Non-blocking operations keep threads compute-bound, not I/O-bound.

Dedicated connection pools

Persistent connections per thread-node pair. Zero setup/teardown overhead per request.



Parallelism without contention. Each thread has its own connection pool to its own node. Threads never compete for the same resources.

DB Calls with Deterministic Routing

Strong consistency without centralized coordination

Raft consensus

Serialized transaction ordering across replicas. No central coordinator. Correctness under partial network failures.

Geo-partitioning

Colocate data near clients. Reduce read/write latency by placing partitions in the same zone as the application.

MVCC + Serializable Isolation

Concurrent workloads without sacrificing consistency. Safe retries via transactional savepoints.

Automatic Failover

Node outages handled transparently. No manual reconfiguration. Replicas promote automatically.

Modern DB + deterministic routing = routing optimizes for probabilistic leaseholder locality with strong ACID guarantees. Not hard node ownership — optimized affinity. (Locality-aware placement: Spanner, OSDI 2012.)

Kubernetes tuning that actually matters

Default settings will kill your latency

StatefulSets for CockroachDB

Stable pod identities, persistent volumes, predictable startup order. Essential for Raft quorum.

Node affinity policies

Colocate compute pods with their database nodes. Minimize network hops between application and storage.

HPA on custom metrics

Scale on P95 latency and queue depth, not just CPU. Default CPU-based scaling misses latency problems.

Resource limits that don't throttle

CPU throttling is the silent killer. Set requests = limits for guaranteed QoS. No burstable for latency-critical pods.

requests = limits = Guaranteed QoS

```
resources:
requests: { cpu: 1, mem: 1Gi }
  limits: { cpu: 1, mem: 1Gi }
```

Results

Benchmarked across 10M+ operations in a controlled 3-node cluster

Metric	Baseline	Optimized	Improvement
P50 latency	18ms	**8ms**	**56% ↓**
P95 latency	34ms	**14ms**	**59% ↓**
P99 latency	47ms	**19ms**	**59% ↓**
Throughput (TPS)	4,200	**8,900**	**2.1x ↑**
Failover recovery	45s	**12s**	**73% ↓**
CPU utilization	35%	**71%**	**2x ↑**

2.1x throughput. 59% P99 reduction. 73% faster failover. Benchmark: 3-node CockroachDB (RF=3) · c6i.2xlarge · EKS · 70/30 read-write · 256 concurrent · 10M ops · uniform keys · 60s warmup. Same infra, only routing and execution changed.

Beyond CockroachDB

The patterns work on any distributed database

CockroachDB

Range-based geo-partitioning + Raft

Hash to range leaseholder node

DynamoDB

Partition key determines shard placement

Hash to partition key = natural affinity

TiDB

Placement rules + TiKV region scheduling

Hash to TiKV store hosting the region

YugabyteDB

Tablet-based sharding + Raft

Hash to tablet leader node

The routing principle is universal. If your database shards by key, you can route by key. The parallelization and K8s tuning patterns transfer directly.

Learnings...

Operational lessons from production

LESSON 1

Deterministic routing needs fallback logic

Hash-to-node coupling is rigid. When a node goes down, all its transactions fail. We added replication-aware redirection and temporary queuing.

Predictability without resilience is fragile.

LESSON 2

Parallelism without isolation makes things worse

First version shared connection pools across threads. Contention was worse than single-threaded. Dedicated pools per thread-node pair fixed it.

More threads ≠ more throughput. Isolation does.

LESSON 3

Kubernetes defaults are latency traps

CPU throttling added 15ms of jitter. Burstable QoS caused evictions during peak. Setting requests = limits eliminated both.

Tune K8s for latency, not just availability.

Where this approach breaks down

Limitations and failure modes worth knowing

Hot partitions

A skewed key (one hot tenant, one viral merchant) overloads its node.

Deterministic routing can't spread a single hot key.

Moving leaseholders

Rebalancing and failover relocate leaseholders at runtime.

The hashed target can be stale — routing must tolerate misses.

Cross-key transactions

Multi-key writes still span nodes and need coordination.

Locality helps single-key paths most, not distributed commits.

Resharding overhead

Topology changes trigger remapping and key-range migration.

Adaptive rebalancing adds real operational complexity.

Know the boundaries. Deterministic routing optimizes the common path — it doesn't replace fallback logic, distributed-transaction coordination, or capacity planning for skew.

What’s next

Planned enhancements

Adaptive rebalancing

When traffic patterns shift, automatically redistribute key ranges without downtime. Hot keys get their own node.

Multi-region deterministic routing

Extend key hashing across regions. Route to the nearest node that has the data, not just any node in the cluster.

Automated failover with routing awareness

When a node fails, the routing layer redirects to the replica that already has the data — not a random healthy node.

Observability integration

Per-key latency attribution. Know which keys are hot, which nodes are slow, and why — before the pager fires.

Each enhancement preserves the core principle: route to the node most likely to hold the data. Everything else is refinement.

Takeaway

One thing to do Monday morning

If your database shards by key,
your application should route by key.

Next Set of Action Items

01 Check your database's partitioning strategy. What key are you sharding on?

02 Trace 100 transactions. How many hit the node that owns their data?

03 Set K8s resource requests = limits on your latency-critical pods.

04 Measure P95 and P99 before and after. The numbers will surprise you.

05 Instrument node_hit_ratio — track the % of requests landing on the correct node.

That's the entire roadmap. Start there.

Thank You

Or come find me by the snacks.

Ritvik Pandya

Engineering Manager

linkedin.com/in/ritvik-pandya

Scan to connect