

System Design & High-Scale Distributed Architecture Cheat Sheet

Latency numbers, consistent hashing, CAP/PACELC theorems, sharding, and back-of-the-envelope capacity math.

EXECUTIVE SUMMARY

Complete architectural reference for designing high-availability distributed systems: capacity estimation formulas, consistent hashing with virtual nodes, read/write replication quorums, rate-limiting algorithms, and multi-tier caching strategies.

KEY EQUATIONS & INVARIANTS

* Strict Quorum Overlap Condition:

LaTeX: $W + R > N \quad \text{and} \quad W > \frac{N}{2}$

Note: Ensures every read quorum R intersects with the latest write quorum W across N replicas.

* Little's Law (Concurrency & Throughput):

LaTeX: $L = \lambda \cdot W$

Note: Average concurrent requests in flight (L) equals arrival rate (λ RPS) times mean latency (W seconds).

* Composite SLA Availability (Serial vs Parallel):

LaTeX: $A_{\text{serial}} = \prod_{i=1}^k A_i, \quad A_{\text{parallel}} = 1 - \prod_{i=1}^k (1 - A_i)$

Note: Two 99.9% services in series yield 99.8% SLA; two 99.9% active-active replicas yield 99.9999%.

1. Back-of-the-Envelope Latency & Capacity Numbers

Quickly convert Daily Active Users (DAU) into Queries Per Second (QPS) using $\frac{\text{DAU}}{\text{day}} \approx 86,400 \text{ s} \approx 10^5 \text{ s}$. Always provision for Peak QPS ($2 \times$ to $5 \times$ average QPS).

Operation / Tier | Typical Latency | Throughput / Bandwidth | Design Implication

L1 / L2 CPU Cache Reference | 0.5 ns - 4 ns | 1+ TB/s | Keep hot loops cache-line aligned (64B)

Main Memory (DRAM) Read | 80 ns - 100 ns | 50+ GB/s | Redis / Memcached in-memory state

NVMe SSD Random Read (4KB) | 16 us - 50 us | 3 - 7 GB/s | WAL append & B+Tree / LSM page lookups

Intra-Region Datacenter RTT | 0.5 ms | 10 - 100 Gbps | Synchronous quorum replication boundary

Cross-Continent WAN RTT (US-EU) | 75 ms - 150 ms | Variable | Requires async replication + edge CDN

[PYTHON] Consistent Hashing Ring with Virtual Nodes

```
import bisect
import hashlib

class ConsistentHashRing:
    def __init__(self, nodes: list[str], vnodes: int = 128) -> None:
        self.vnodes = vnodes
        self.ring: list[tuple[int, str]] = []
        for node in nodes:
            self.add_node(node)

    def _hash(self, key: str) -> int:
        return int(hashlib.md5(key.encode()).hexdigest(), 16)
```

```
def add_node(self, node: str) -> None:
    for i in range(self.vnodes):
        h = self._hash(f'{node}#vnode{i}')
        bisect.insort(self.ring, (h, node))

def get_node(self, key: str) -> str:
    h = self._hash(key)
    idx = bisect.bisect_left(self.ring, (h, ''))
    return self.ring[idx % len(self.ring)][1]
```

2. Caching Topologies & Distributed Rate Limiting

Prevent ****Cache Stampede (Thundering Herd)**** on hot-key expiration using request coalescing (singleflight), probabilistic early expiration (XFetch), or lease tokens.

Pattern | Write Path | Read Path | Tradeoff

Cache-Aside | App writes DB, invalidates cache | Read cache -> miss -> read DB -> populate | Resilient to cache failure; risk of stale read race
 Write-Through | App writes cache + DB synchronously | Always hits fresh cache | Higher write latency; zero stale reads
 Write-Behind (Async) | App writes cache; worker flushes batch to DB | Always hits cache | Highest write throughput; risk of data loss on crash

COMMON PRODUCTION & INTERVIEW PITFALLS

- Using modulo hashing (`hash(key) % N`) for cache sharding causes $(N-1)/N$ keys to relocate when a single node scales.
- Relying on wall-clock timestamps (`time.time()`) for distributed ordering across servers with NTP skew causes silent data loss under Last-Write-Wins (LWW).
- Retrying failed RPCs immediately without exponential backoff and full jitter amplifies cascading outages.

FREQUENTLY ASKED QUESTIONS

Q: What is the PACELC theorem and how does it extend CAP?

A: CAP only describes behavior during a network Partition (choose Availability or Consistency). PACELC adds that Else (when the network is healthy), a distributed database must still choose between Latency (L) and Consistency (C).

Q: When should I choose an LSM-Tree (Cassandra/RocksDB) over a B+Tree (PostgreSQL)?

A: Choose LSM-Trees for write-heavy ingestion workloads (sequential memtable + SSTable appends avoid random disk I/O). Choose B+Trees when low-latency point reads and transactional range scans dominate.