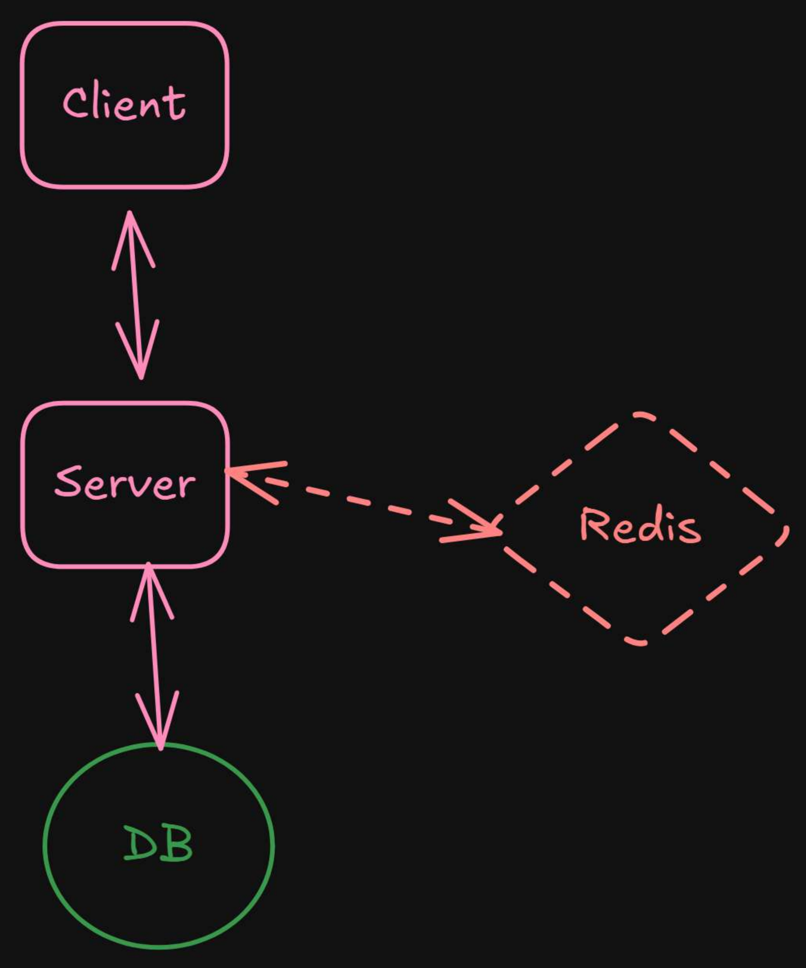
Why Redis Exists

Imagine your app shows the user's profile.
Every time a user opens the app → you fetch from database.

Now your app grows to 1 million users.
Suddenly:
-> your DB gets hammered
-> queries take 300–500 ms
-> users complain the app is slow



But wait...
Most users are requesting the same data again and again.

What if you could store the result temporarily in RAM,
so the next time someone asks for it,
you return it in 0.5 milliseconds instead of 500 ms?

That "super-fast temporary memory layer" is Redis.

What is Redis?

Redis (Remote Dictionary Server) is an:
In-memory, extremely fast key–value data store
used as:

-> Cache
-> Message broker
-> Distributed locking system
-> Stream processing engine

It is popular because it is insanely fast, supports rich data structures, and solves real performance bottlenecks in modern systems.

There is one major limitation: durability.
Because Redis operates primarily from RAM, it doesn't give the same strong "once I commit, it's on disk forever" guarantee that traditional databases give. It does provide persistence mechanisms, such as the Append-Only File (AOF), which greatly reduce the risk of losing data — but they don't match the durability of fully disk-based systems.
This is a conscious design decision:
Redis prioritizes speed over absolute safety.

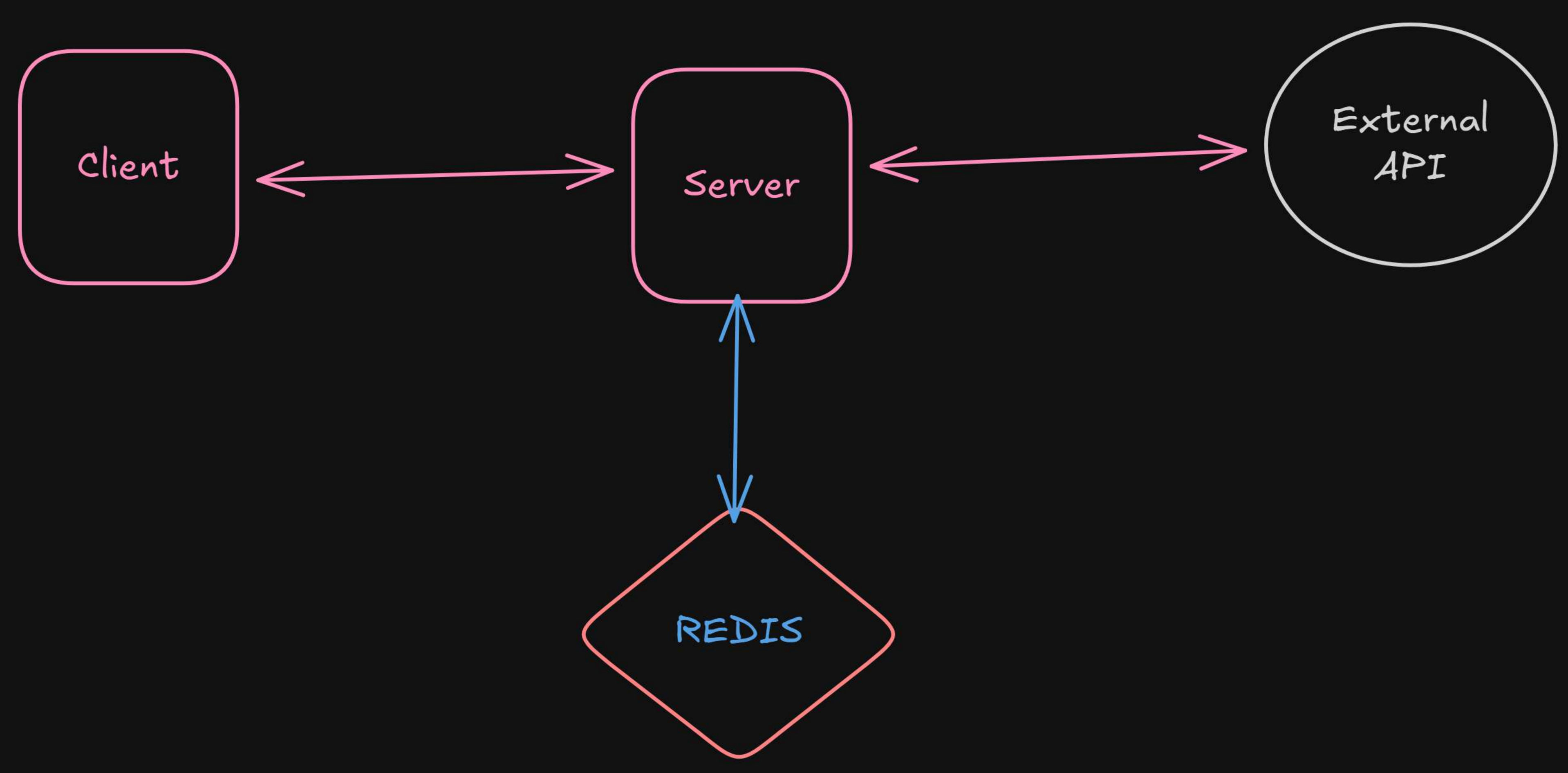
If you really need Redis-like behavior with stronger durability, there are cloud variants (like AWS MemoryDB) that add disk-backed persistence while accepting a small performance hit.

Why Is Redis So Fast?

In-memory storage
-> Everything is stored in RAM → extremely fast.
Single-threaded event loop
-> No locks
-> No race conditions
-> No thread switching overhead
Simple data structures
-> Redis uses highly optimized C-level implementations (sds strings, dict, skiplist, intset, ziplist/quicklist, etc.)

Together → Redis is usually 100x+ faster than databases like MySQL or MongoDB.

DEMO



Core Redis Data Structures

Redis goes beyond simple strings.
It offers a rich collection of structures,
each optimized for a particular use case:

Strings (basic values, counters)

Hashes (like JSON objects or dictionaries)

Lists (ideal for queues)

Sets (unique collections)

Sorted Sets (think priority queues or leaderboards)

Time series structures (for monitoring metrics or events)

Geospatial indexes (location-based queries)

Bloom Filters (space-efficient membership tests with possible false positives)

And Redis isn't limited to storing data —
it also supports communication patterns:

-> Pub/Sub (real-time messaging)

-> Streams (lightweight log-based messaging, similar to Kafka-lite)

Because of these, Redis can sometimes
replace more complex systems like Kafka, SNS, or SQS
for smaller or simpler workloads.

Redis Deployment Models / Configurations

Redis can operate in several configurations
depending on your
durability, performance, and scaling needs.
At its core, Redis stays lightweight
— even its clustering approach is intentionally minimal so
that you decide how to distribute your data.

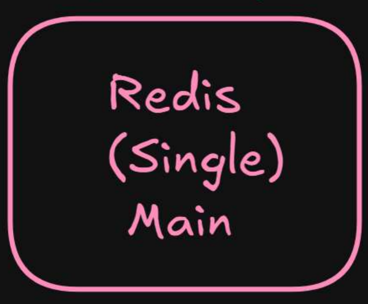
1. Single-Node Setup (The simplest form)

A standalone Redis server. One machine, no replicas.

All reads/writes go to one server.

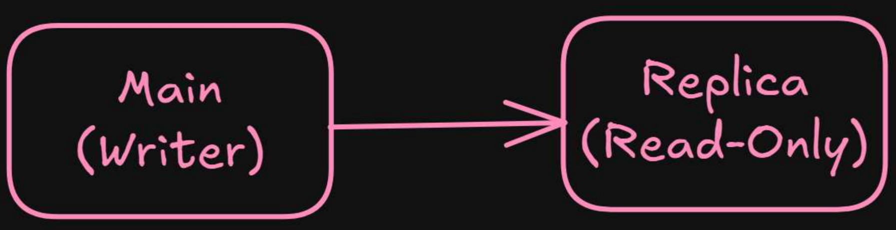
Easiest to run.

No failover.



2. Replication / High Availability Setup

Here, a main instance is paired with replicas.
Replicas synchronize from the main node and
can take over during failures.



Main handles writes.

Replicas can take traffic for reads.

Provides fault tolerance but not horizontal scaling for writes.

3. Redis Cluster Setup

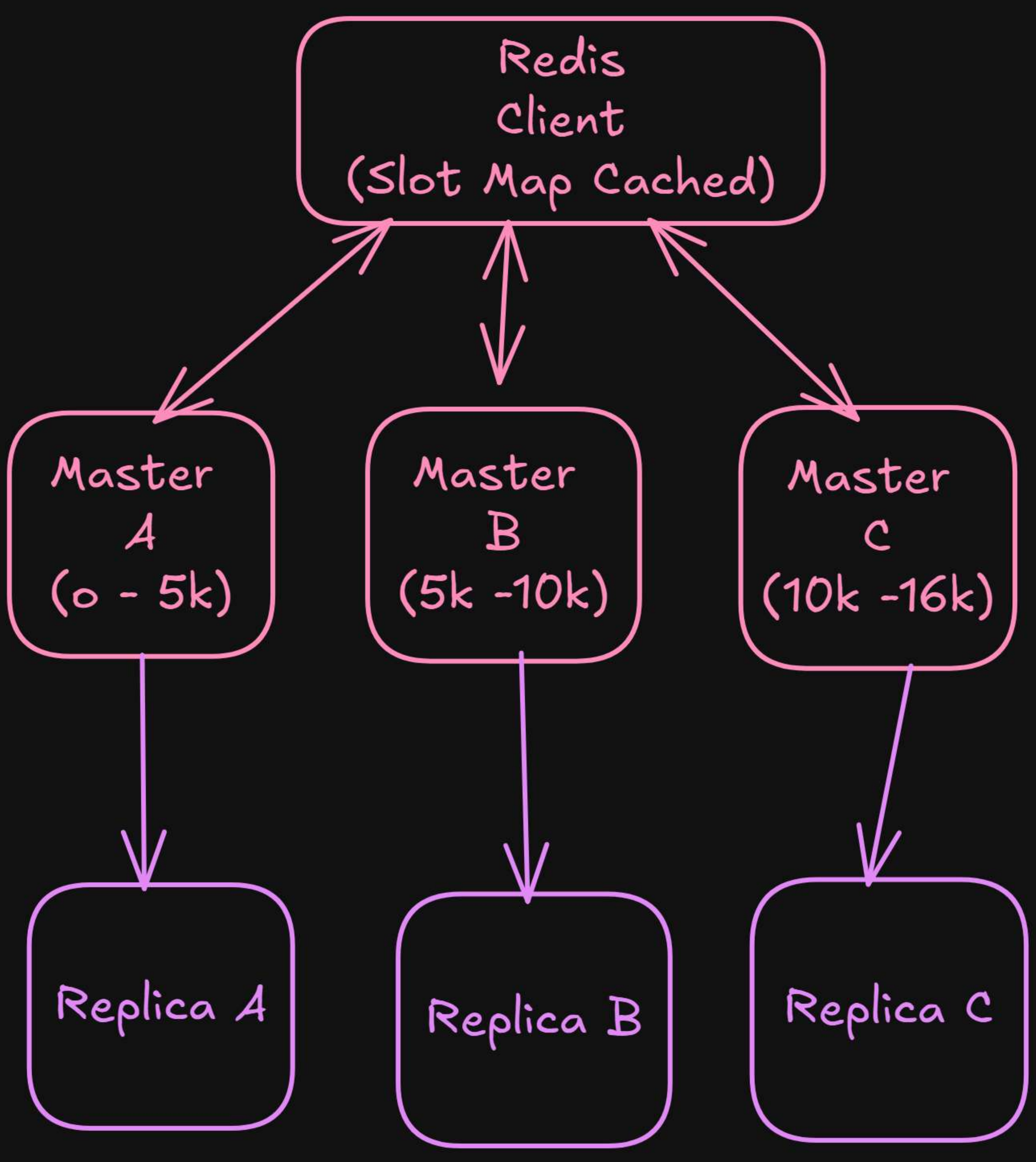
Redis Cluster partitions data across multiple masters using hash slots.
Each master can also have its own replica.

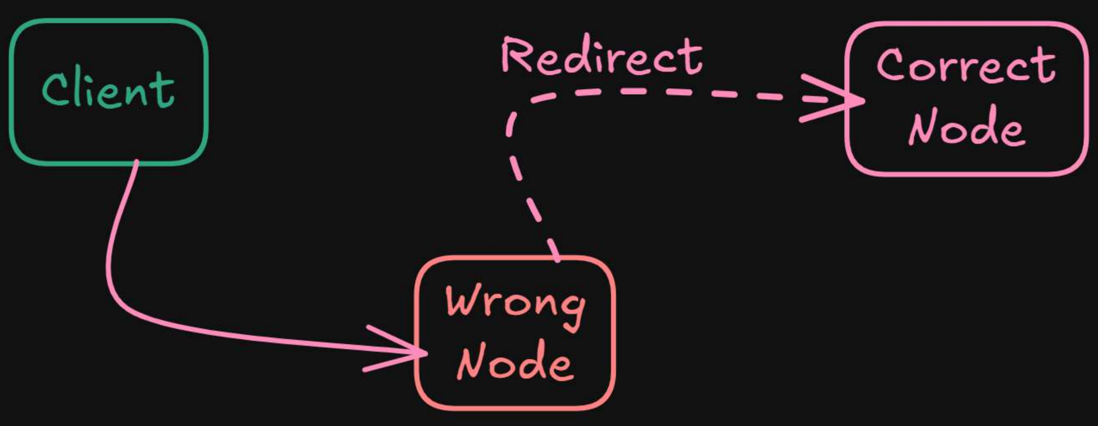
1. Keyspace split
   Redis divides all data into 16,384 hash slots.

2. Slot hashing
   Key slot = CRC16(key) % 16384.

3. Masters own slots
   Each master node is assigned a range of slots.
   Replicas exist for HA.

4. Cluster gossip
   Each node has partial knowledge of other nodes through a
   lightweight gossip mechanism.
   This enables limited rerouting when a request lands on the wrong master:





5. Client discovers topology
   Client runs CLUSTER SLOTS / CLUSTER SHARDS.
   Builds a local slot → node map.

6. Client-side routing
   Client computes slot for each key.
   Sends command directly to the correct master.
   Redis has no proxy.

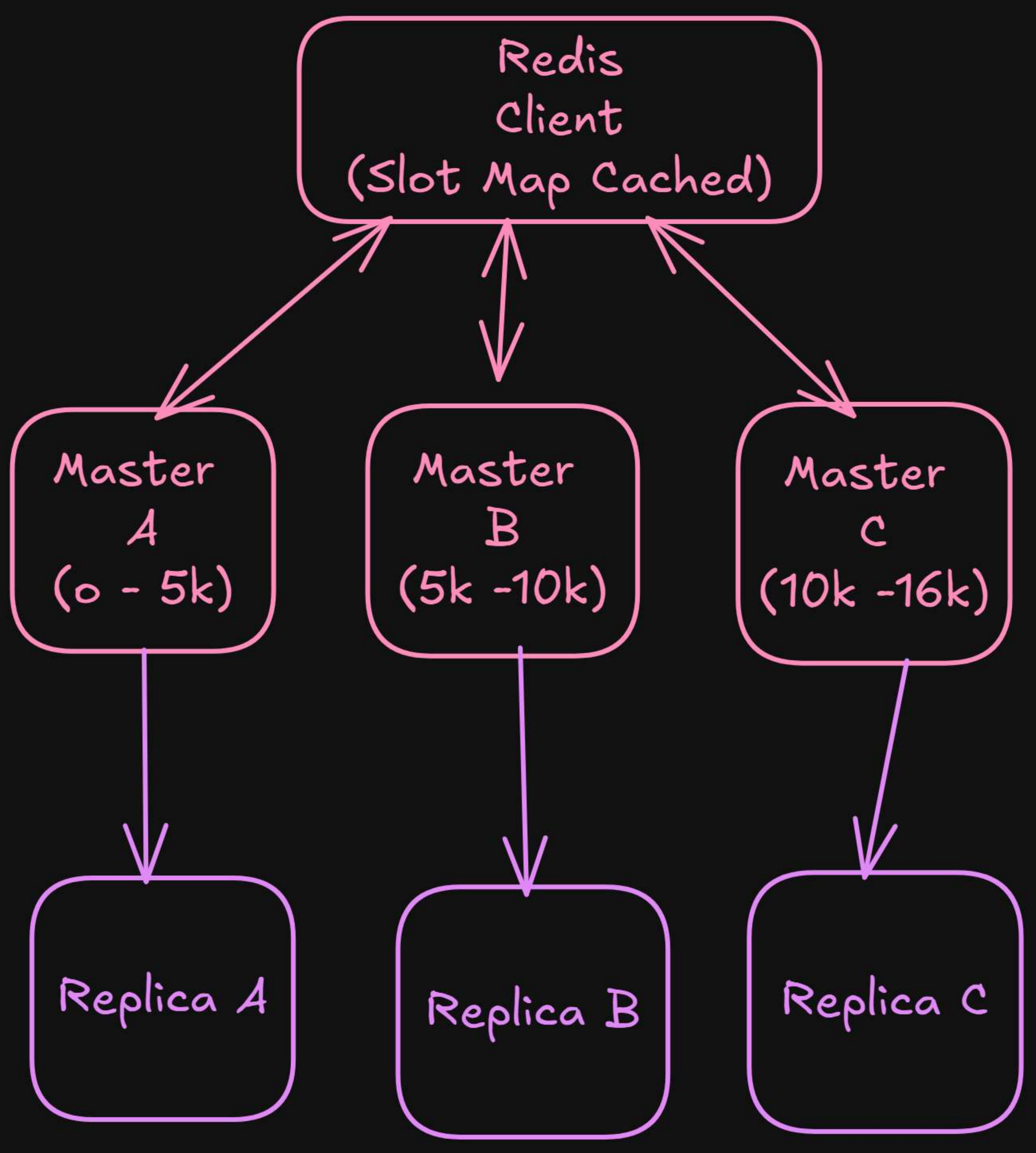
7. Failover
   If master dies → replicas vote → one is promoted.
   Cluster updates slot ownership.
   Clients get MOVED and refresh routing.

8. Writes go to master
   Masters handle writes.
   Replicas sync asynchronously.

9. If something changes (failover, rebalancing), a node will respond:

MOVED <slot> <correct-node>

And the client refreshes its internal slot map.



Important Limitation: Redis Cluster Is Very Simple

Redis Cluster is intentionally minimalistic:

Good for:
- Large datasets that don't fit on one node
- High availability
- Horizontal read/write scaling (across masters)

Not good at:
- Multi-key operations across different slots
- Complex joins/transactions
- Moving lots of data automatically

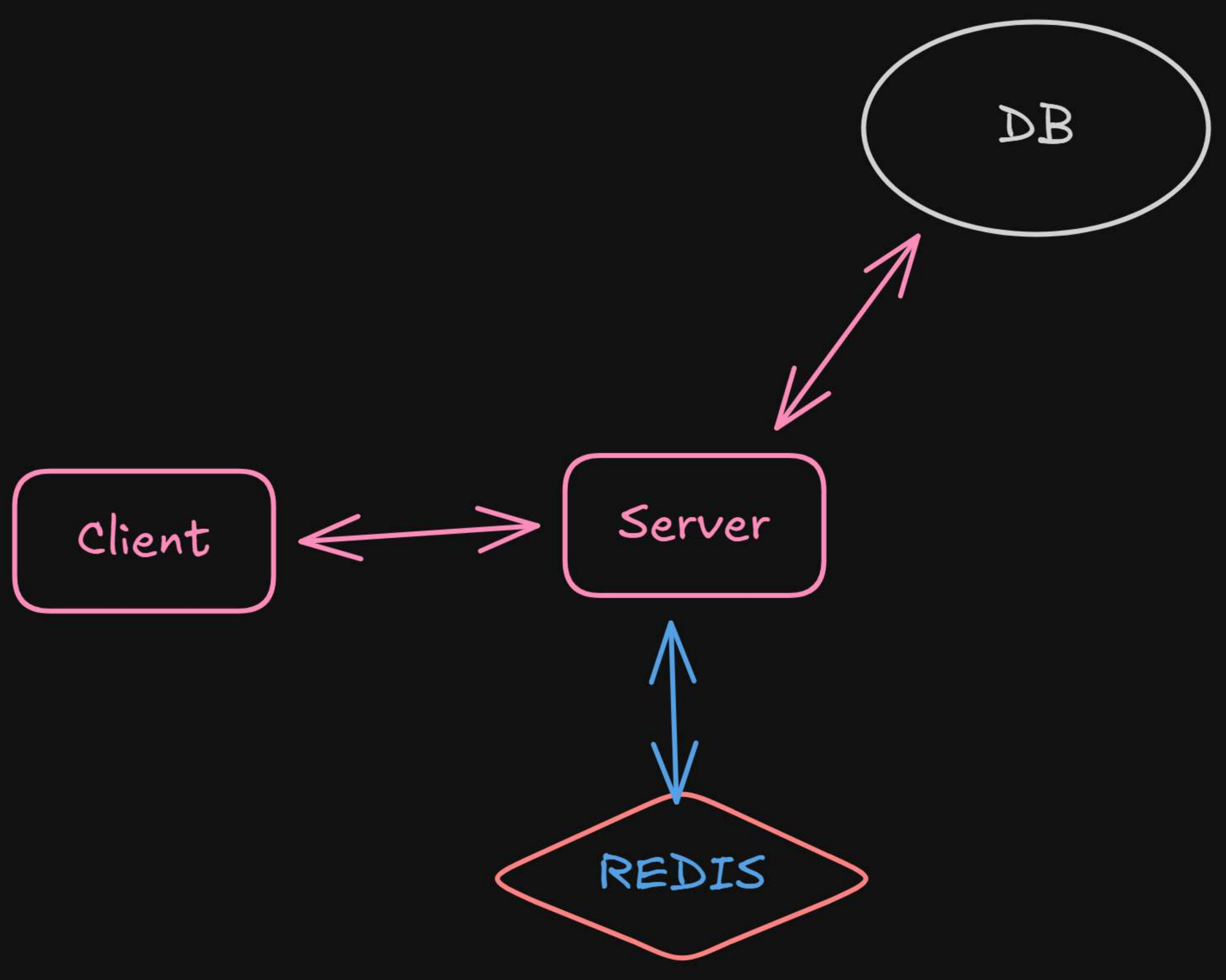
Redis expects all keys involved in a request to live on the same node.
That means the way you design key names determines scalability.

Redis as a Cache

Redis is commonly deployed as a fast in-memory cache to reduce load on primary databases and speed up responses.

Each Redis key represents a cached item.
Example: a user profile stored under user:42 containing fields like name, email, and lastLogin.

Redis Cluster can spread keys across multiple nodes, allowing the cache to scale horizontally.
Need more capacity or throughput? → Add more nodes.



Cached user profiles typically have a TTL (time to live):

Redis ensures expired profiles are never returned.

TTL helps Redis automatically evict old or unused profiles when memory is constrained.

Caching user profiles in Redis does not fix the "hot key" issue:

If one user profile (e.g., user:42) is requested extremely frequently, the node storing that key may become overloaded.

This challenge exists in all distributed caches (Redis, Memcached, DynamoDB, etc.).

How to Solve / Mitigate Hot Keys in Redis

A hot key is a single Redis key that receives massive,
disproportionate traffic compared to others.
This can overload the node that owns that slot.

1. Add Replicas + Enable Replica Reads

2. Cache-Breaking Using Key Sharding (Randomized Keys)
   Instead of storing one hot key, store N copies of it
   under different keys:
   - user:42:profile:1
   - user:42:profile:2
   - user:42:profile:3
   - user:42:profile:4

   Requests spread across multiple slots → multiple cluster nodes.

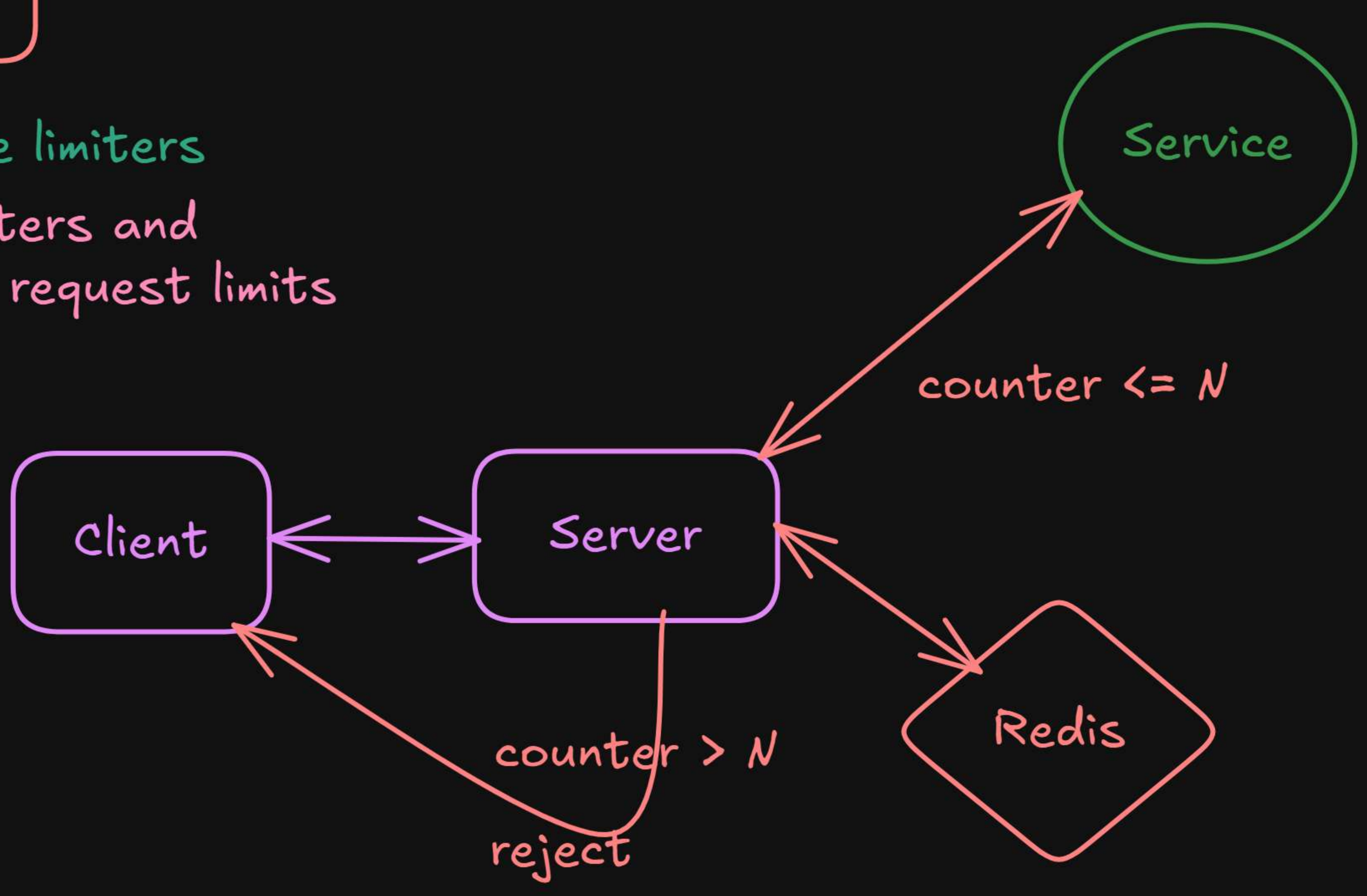
3. Local In-App Cache (L1 Cache)
   Add a small in-memory cache inside your
   application (e.g., Node.js memory, Java LRU map):
   - First check local cache
   - Only hit Redis if not found
   - TTL can be very short (e.g., 50–200ms)

   Benefit:
   Thousands of requests hit your application memory instead of RediS

Redis for Rate Limiting

Redis is perfect for implementing rate limiters

Because Redis supports atomic counters and
time-based expiration, it can enforce request limits
with very little overhead.

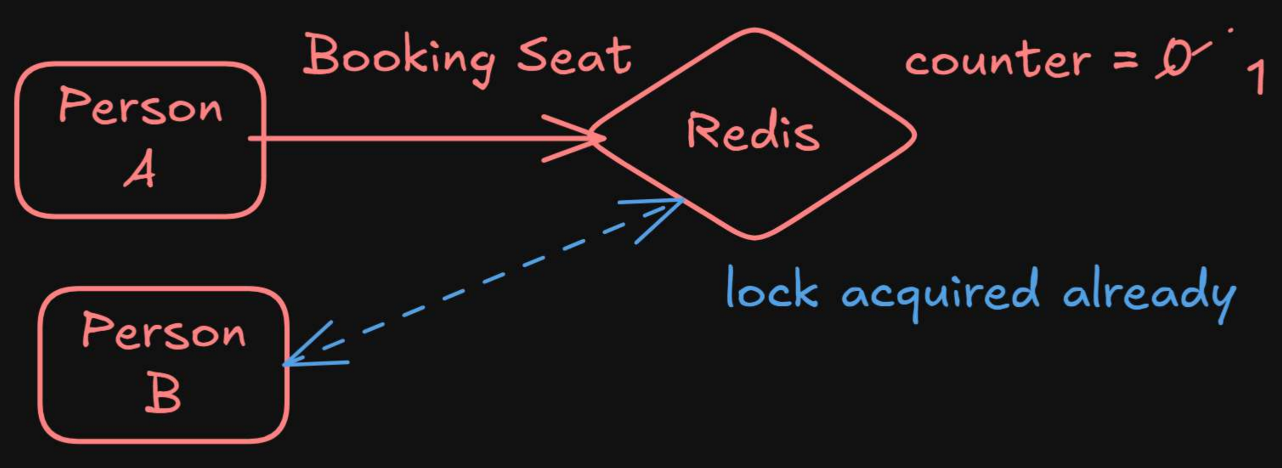


You define a time window (ttl seconds) and a request limit (N requests).
For each incoming request:

Increase a counter for that user/API/token
If the count exceeds N → reject or delay
Apply a TTL so the counter automatically resets after ttl seconds

Redis as a Distributed Lock

A minimal lock can be implemented using an atomic counter



-> Redis can coordinate access to shared resources

Redis can function as a simple mechanism to ensure
that only one process performs a sensitive operation at a time
—useful in scenarios like ticket reservation or preventing
two users from updating the same record simultaneously.

-> Only use a Redis lock when your main database cannot
guarantee consistency

If your primary datastore already enforces correctness,
adding a distributed lock may create extra complexity
and new edge cases.

A lock key starts at zero.
When a process tries to acquire the lock:
It performs INCR lock_key.
If the result is 1, it successfully holds the lock.
If the result is > 1, another process already owns it.
Apply a TTL so the lock disappears automatically if something crashes.
When done, the process deletes the key (DEL lock_key).

This acts like a small shared counter where only the first incrementer wins.