📘 Caching with Redis

🔰 Introduction to Redis

Redis (REmote DIctionary Server) is:

- An **in-memory data store** — meaning all data is stored in RAM for ultra-fast access.
- Often used as a **cache** because of its low latency.
- Also works as a **NoSQL database**, **message broker**, and for **real-time analytics**.

Key Features:

- Lightning-fast performance (sub-millisecond response)
- Supports rich data types (beyond just strings)
- Persistent options (AOF and RDB)
- Built-in Pub/Sub, Lua scripting, clustering, and replication

---

🧩 Why Use Caching?

Imagine a popular e-commerce site:

- Millions of users visit the product page
- Without caching, every visit hits the database
- This causes bottlenecks, latency, and cost spikes

Benefits of Caching:

- ⏱️ **Improves performance** — repeated requests are served from memory

- 📉 **Reduces backend/database load**
- 💵 **Saves cost** — fewer database reads
- 🔄 **Ensures scalability** during traffic spikes

📦 Caching Strategies Explained

🕑 1. Time-To-Live (TTL)

Each cache entry should **expire automatically** to avoid stale data.

SET product:123 '{"name": "Laptop"}' EX 300 # Expires in 5 minutes

- Useful for data that changes frequently (e.g., prices, stock)
- Prevents serving outdated info

🧹 2. Eviction Policies

When Redis reaches memory limit, it **evicts keys** based on configured policy.

Common policies:

- `noeviction` — new writes fail when memory is full
- `allkeys-lru` — evict Least Recently Used key
- `allkeys-lfu` — evict Least Frequently Used
- `volatile-lru` — only evict LRU among keys with TTL
- `volatile-ttl` — evict the one with the nearest TTL

Set in `redis.conf` or via CLI:

CONFIG SET maxmemory-policy allkeys-lru

🔁 3. Cache Invalidation Strategies

When data in DB changes, **invalidate** or **refresh** the cache.

a) Write-through

- Write to **cache and DB simultaneously**
- Ensures consistency
- Slower on writes

b) Write-behind

- Write to cache, and **sync to DB later**
- Good for performance
- Risk of data loss on crash

c) Cache-aside (Lazy loading)

- App reads from cache first
- On miss → load from DB and populate cache
- Most common pattern

Python Example (Cache-aside):

```
def get_user(user_id):
    data = redis.get(f"user:{user_id}")
    if data:
        return data
    user = db.query(user_id)
    redis.set(f"user:{user_id}", user, ex=300)
    return user
```

🧱 Redis Data Structures

Redis is not just key-value strings — it supports advanced structures:

🔤 1. Strings

- Basic key-value pair
- Fastest and simplest

```
SET token:abc123 "user_id:42"
GET token:abc123
INCR views:homepage
```

Use cases: session tokens, counters, flags

📝 2. Hashes

- Store related fields under one key
- Like a mini JSON object

```
HSET user:42 name "Alice" age 30
HGET user:42 name
HGETALL user:42
```

Use cases: user profiles, product metadata

📋 3. Lists

- Ordered, can push/pop from both ends
- Behaves like queue or stack

```
LPUSH tasks "email_signup"
RPUSH tasks "send_notification"
LPOP tasks
```

Use cases: task queues, recent activity logs

🔢 4. Sets

- Unordered, no duplicates
- Fast operations: union, intersection

```
SADD online_users "user42"
SISMEMBER online_users "user42"
SMEMBERS online_users
```

Use cases: tracking active users, tags

🧮 5. Sorted Sets

- Like sets but with scores → sorted results

```
ZADD leaderboard 500 "Alice"
ZADD leaderboard 1000 "Bob"
ZRANGE leaderboard 0 -1 WITHSCORES
```

Use cases: leaderboards, priority queues, ranking

🧠 6. Bitmaps

- Memory-efficient storage for true/false values
- Track presence, activity

```
SETBIT users_online 1001 1
GETBIT users_online 1001
```

Use cases: tracking daily login, feature flags

📊 7. HyperLogLog

- Approximate unique count (very memory efficient)

```
PFADD visits user1 user2 user3
PFCOUNT visits
```

Use cases: analytics (daily active users), uniqueness estimation

---

🗺️ 8. Geospatial Indexes

- Store location data and query by radius

```
GEOADD places 13.361389 38.115556 "Palermo"
GEORADIUS places 15 37 200 km
```

Use cases: store locator, delivery zones

🔀 9. Streams

- Log-based data structure (append-only)
- For real-time pipelines, event sourcing

```
XADD mystream * event login user alice
XRANGE mystream - +
```

Use cases: logs, real-time messaging, async data pipelines

💼 Practical Integration: FastAPI + Redis

✅ Scenario: Cache an API Response

1. Create an API that returns DB data

2. On first call → fetch from DB, cache it in Redis
3. Next calls → return from Redis (fast)

```
from fastapi import FastAPI
import redis
import time

app = FastAPI()
r = redis.Redis(host="localhost", port=6379)

@app.get("/product")
def get_product():
    cached = r.get("product:123")
    if cached:
        return {"source": "cache", "data": cached.decode()}

    time.sleep(2)  # simulate slow DB
    product = '{"name": "Laptop", "price": "$999"}'
    r.set("product:123", product, ex=60)
    return {"source": "db", "data": product}
```

⚙️ Benchmarking Performance

Use `ab` (Apache Bench) or `wrk`:

```
ab -n 100 -c 10 http://localhost:8000/product
```

Expected:

- First call: ~2s (DB)
- Next 99: ~1-5ms (Redis cache)

✅ Best Practices with Redis Caching

- Use meaningful keys: `user:42:profile`, `cache:product:123`
- Always set TTL to avoid stale data and OOM errors

- Choose the right eviction policy (`allkeys-lru` is a good default)
- Monitor Redis usage: `INFO`, `MONITOR`, Redis CLI
- Avoid large keys or values → Redis is optimized for small, fast data
- Namespace keys using prefixes (`auth:`, `cache:`, `session:`)

🧠 Summary

Concept	Description
Redis as cache	Fast access layer for frequently accessed data
TTL	Auto-expire stale cache
Eviction policies	Manage memory by removing old/unused data
Data structures	Strings, Hashes, Lists, Sets, Sorted Sets, Bitmaps, etc.
Cache-aside strategy	Most common, fetch & populate only on cache miss
Practical benefit	Reduces DB load, boosts response time

Would you like the next section to be **visual architecture diagrams**, **quiz questions for students**, or **deployment practices** (e.g., Redis in Docker or AWS ElastiCache)?

Explanation for Demo 1: Strings

- **Concept:** This demo introduces the most fundamental Redis data structure: the **String**. Think of it as the simplest key-value pair. You give Redis a key (like `session:user:123`) and it stores a single string value.
- **Use Case 1 (Session Store):** We show how to store a user's session token. The key command here is `SETEX`, which sets a value but also attaches an **expiration time (TTL)**. This is the perfect pattern for sessions because you don't have to manually clean up old data; Redis removes the key automatically after the timer runs out. The demo proves this by showing the key exists initially but is `None` (gone) after waiting 11 seconds.
- **Use Case 2 (Counter):** We demonstrate the `INCR` command. Even though the value is a string, Redis is smart enough to treat it as a number if possible. `INCR` is an **atomic operation**, meaning even if a thousand users hit a webpage at the same exact moment, the view counter will be perfectly accurate. This makes it ideal for any kind of real-time counting.

Explanation for Demo 2: Lists

- **Concept:** This demo showcases **Lists**, which are ordered collections of strings, much like a Python list or an array. Unlike Sets, you can have duplicate values. They are optimized for adding or removing items from the beginning or the end.
- **Use Case (Task Queue):** The classic use case for a List is a simple message or task queue. The demo simulates this with a "producer" and a "worker":
    - The **producer** uses `LPUSH` to add new jobs to the left-hand side (the start) of the list.
    - The **worker** uses `RPOP` to pull the oldest job from the right-hand side (the end) of the list.
    - This `LPUSH`/`RPOP` combination creates a **FIFO (First-In, First-Out)** queue, which is a very common and fair way to process background jobs.

Explanation for Demo 3: Hashes

- **Concept:** This demo explains **Hashes**. A Hash allows you to store a "mini dictionary" or an object within a single Redis key. Instead of storing a user's profile across many string keys (`user:77:username`, `user:77:email`, etc.), you can store it all under one key (`user:profile:77`) with multiple fields inside.
- **Use Case (User Profile):** This is the perfect way to model an object. The code shows how `HSET` can create a profile with fields like `username`, `email`, and `followers`. `HGETALL` retrieves the entire object, which is very efficient. The demo also shows `HINCRBY`,

which atomically increments a numeric field inside the hash, perfect for updating things like follower counts or game scores.

Explanation for Demo 4: Sets

- **Concept:** This demo introduces **Sets**, which are unordered collections of *unique* strings. The "unique" part is key—if you try to add the same item twice, it's simply ignored.
- **Use Case (Tagging System):** Sets are ideal for managing tags. A blog post can be tagged with "redis" and "caching".
- **The Power of Set Operations:** The most important part of this demo is showing the built-in mathematical operations that Redis can perform at very high speed:
    - `SINTER` (Intersection): Finds the common tags between two posts. This is great for finding related content.
    - `SDIFF` (Difference): Finds tags that are in the first post but not the second.
    - `SUNION` (Union): Creates a combined list of every unique tag used across both posts. Performing these operations inside Redis is far more efficient than fetching two lists and comparing them in your Python code.

Explanation for Demo 5: Sorted Sets (ZSETs)

- **Concept:** This demo explains **Sorted Sets**, which are the "superpower" data structure in Redis. They are like Sets (unique members), but each member has an associated **score**. Redis uses this score to keep the entire collection sorted at all times.
- **Use Case (Gaming Leaderboard):** This is the classic example. A player's ID is the member, and their point total is the score.
- **Key Functionality:** The demo shows how easy it is to manage a leaderboard:
    - `ZADD` is used to both add and update scores. Redis automatically re-sorts the leaderboard when a score changes.
    - `ZREVRANGE` is used to get the top players (it gets a range in *reverse* order, from highest score to lowest).
    - `ZREVRANK` and `ZSCORE` allow you to instantly find a specific player's rank or score without having to scan the whole list.

Explanation for Demo 6: Bitmaps

- **Concept:** This demo showcases **Bitmaps**. A Bitmap isn't a separate data type but rather a set of bit-level commands that operate on a normal Redis String. It's an incredibly memory-efficient way to track boolean (true/false) information for millions of

items.
- **Use Case (Daily Activity Tracker):** We track which users logged in on a given day. The user's ID corresponds to a bit's **position (offset)** in the string. If user 50 logs in, we set the 50th bit to `1`.
- **Key Benefits:**
    - **Memory:** As noted, you can track millions of users' daily status in just a few megabytes.
    - **Analytics:** `BITCOUNT` instantly tells you the total number of unique active users for that day. `BITOP` lets you perform logical operations, like finding the `AND` between two days to calculate user retention—a very powerful feature for analytics.

Explanation for Demo 7: HyperLogLog

- **Concept:** This demo introduces **HyperLogLog (HLL)**, a probabilistic data structure. Its one job is to **estimate the number of unique items** in a set. It trades perfect accuracy for extreme memory efficiency.
- **Use Case (Unique Search Counter):** We want to know how many *distinct* search terms were used, not the total number of searches. Many users might search for "redis tutorial".
- **Key Trade-off:** HLL can count billions of unique items and will still only consume about 12 KB of memory. The demo shows that its count is very close to the actual unique count. This is perfect for large-scale analytics where a 99% accurate count is good enough (like a "trending topics" feature).

Explanation for Demo 8: Geospatial

- **Concept:** This demo explains Redis's built-in **Geospatial** capabilities. Redis can store and query items based on their longitude and latitude. Under the hood, it uses a Sorted Set with a special encoding called Geohash.
- **Use Case (Nearby Cafe Finder):** We add several cafes in Bengaluru with their coordinates.
- **Key Functionality:**
    - `GEOADD` stores the location data.
    - `GEODIST` calculates the straight-line distance between two points.
    - `GEORADIUSBYMEMBER` is the most powerful command. It allows you to query for all items within a certain radius of another item, making "find near me" features very fast and easy to implement.

Explanation for Demo 9: Streams

- **Concept:** This demo introduces **Streams**, a robust, append-only log data structure. Think of it as a super-powered List designed for high-speed data ingestion and complex consumption patterns.
- **Use Case (Real-time Sensor Log):** This is a perfect use case for Streams. A sensor is constantly publishing new data (temperature, humidity).
- **Key Functionality:**
    - `XADD` adds a new entry to the end of the log. Each entry gets a unique, time-based ID and can hold multiple field-value pairs.
    - `XLEN` checks the size of the log.
    - `XRANGE` allows you to query a historical range of data from the stream (e.g., "show me all readings from the last hour").
    - `XREAD` is used to "listen" for new data as it arrives, making it ideal for real-time processing services.

Checking the speed of requests with and without redis

Prerequisites

1. **Python 3:** Ensure Python 3 is installed. Verify the version using `python3 --version`.
2. **Redis:** The Redis server must be installed and operational.
3. **Pip:** Pip is required for the installation of Python packages.

Step 1: Project File Creation

Initiate the project structure by creating all necessary files. Open your terminal and execute the following commands:

```
# Create the main project folder and navigate into it

mkdir redis-api-demo

cd redis-api-demo



# Create the sub-folders

mkdir -p api load_test



# --- Create the "Slow" API file first ---
```

```
cat > api/app.py << EOL

import time

import os

from flask import Flask



app = Flask(__name__)



def get_product_from_db(product_id: int):

    """This function simulates a slow database query."""

    print(f"DATABASE HIT: Fetching product {product_id} from the database...")

    time.sleep(2)  # Simulate a 2-second database delay

    return {"id": product_id, "name": f"Product {product_id}", "price": 100.0}



@app.route("/product/<int:product_id>")
```

```
def get_product(product_id):
    """API endpoint to get a product. This version has NO CACHING."""
    data = get_product_from_db(product_id)
    return data


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
EOL

# Create the Python requirements file
cat > api/requirements.txt << EOL
Flask
redis
EOL
```

```
# --- Create the Load Test file ---

cat > load_test/locustfile.py << EOL

from locust import HttpUser, task, between

import random



class APIUser(HttpUser):

    # Wait between 1 and 2 seconds between making requests

    wait_time = between(1, 2)



    @task

    def get_product(self):

        # Pick a random product ID from 1 to 10.

        product_id = random.randint(1, 10)
```

```
        # Group all requests to this endpoint under one name in the statistics report
        self.client.get(f"/product/{product_id}", name="/product/[id]")
EOL

"Project files created successfully."
```

Step 2: Redis Installation and Execution

Ensure Redis is installed and actively running within the same terminal session.

*   **On macOS (using Homebrew):**

    ```bash
    ```

brew install redis

brew services start redis

```

* **On Linux (Ubuntu/Debian):**

```bash

sudo apt-get update

sudo apt-get install redis-server

sudo service redis-server start

```

Validate the Redis service by executing:
```

```bash

redis-cli ping

A successful response will be "PONG."Step 3: Uncached API Benchmarking

Proceed to benchmark the uncached version of the API.

1. **Navigate to the `api` directory:**
   cd api

   **Create and activate a Python virtual environment:**
   python3 -m venv venv

2. source venv/bin/activate
   The terminal prompt will now display `(venv)`.
3. **Install Python packages:**
   pip3 install -r requirements.txt
4. **Run the uncached API:**
   python3 app.py
   This terminal will now display Flask server logs. **Keep this terminal active.**

   **Execute the load test:**
   Open a new, second terminal window.
   # In the NEW terminal:

# Navigate back to your project folder
```

cd path/to/your/redis-api-demo

Install locust

pip3 install locust

Navigate to the load test directory

cd load_test

Start Locust

5. locust -f locustfile.py --host=http://localhost:5000
6. **Analyze the "Slow" results:**
    - Access http://localhost:8089 in your browser.
    - Initiate a test with **10 users** and a spawn rate of **1**.
    - Observe the elevated response times (~2000ms) and reduced requests per second within the Locust UI.
    - Note the recurring "DATABASE HIT" message for each request in the initial API terminal.

Step 4: Caching Integration and Re-Benchmarking

Witness the performance enhancement through caching integration.

1. **Stop the API server:** In the first terminal running the API, press `Ctrl + C`.

   Replace the API code: Substitute the existing `api/app.py` with the "fast" version of the API code. Execute the following command in your terminal (ensure you are in the main `redis-api-demo` directory):
   cat > api/app.py << EOL

   import time

   import redis

   import os

   import json # Import json for serialization

   from flask import Flask

   app = Flask(__name__)

   # Connect to the Redis service - defaults to localhost which is correct

   cache = redis.Redis(host=os.environ.get('REDIS_HOST', 'localhost'), port=6379, decode_responses=True)

```
def get_product_from_db(product_id: int):
    """This function simulates a slow database query."""
    print(f"DATABASE HIT: Fetching product {product_id} from the database...")
    time.sleep(2)  # Simulate a 2-second database delay
    return {"id": product_id, "name": f"Product {product_id}", "price": 100.0}


@app.route("/product/<int:product_id>")
def get_product(product_id):
    """
    API endpoint that first checks the cache.
    If the data is not in the cache, it fetches from the DB and stores it.
    """
    cache_key = f"product:{product_id}"
```

```
# 1. Check cache first

cached_product = cache.get(cache_key)

if cached_product:

    print(f"CACHE HIT: Fetching product {product_id} from cache.")

    return json.loads(cached_product) # Deserialize from JSON string



# 2. If cache miss, get from DB

print(f"CACHE MISS: Product {product_id} not found in cache.")

product_data = get_product_from_db(product_id)



# 3. Store in cache for next time with a 60-second TTL (Time-To-Live)

cache.setex(cache_key, 60, json.dumps(product_data))



return product_data
```

if __name__ == "__main__":

app.run(host="0.0.0.0", port=5000, debug=True)

2. EOL

Run the cached API:

Return to your first terminal. The virtual environment should remain active; if not, reactivate it (`source api/venv/bin/activate`).
Make sure you are in the api directory

cd api

3. python3 app.py
4. **Analyze the "Fast" results:**
    - Revisit the Locust UI at http://localhost:8089 and commence a new test with identical parameters.
    - Observe the drastic reduction in response times to a few milliseconds and the rapid increase in RPS subsequent to the initial cache "warm-up."
    - In your API terminal, confirm that "DATABASE HIT" appears only once per product, followed by a continuous stream of "CACHE HIT" messages.
    -