Assignment: Implement Website Visit Counter

Release Date: 15th February, 2025

Due Date: 23rd February, 2025 EOD

Total Points: 100

Overview

In this assignment, you will implement a **Website Visit Counter** system that keeps track of website **page visits** efficiently. You are required to implement application caching, use Redis for Global Caching, and handle system scaling via Sharding, Batching, etc.

Learning Objectives

By completing this assignment, you will:

- Implement caching strategies for high-performance systems.
- Work with Redis for efficient data storage and retrieval.
- Design a scalable system with sharding, batching etc.
- Understand how consistent hashing is used in Real Systems

Tasks & Scoring

Task 1: Basic Visit Counter (20 points)

Objective:

Implement a basic visit counter that increments every time a user visits a page of the website. Assume every page is identified via a page id.

Requirements:

- Develop a simple backend API (POST `/visits/{page_id}`) that increments a counter on every page load.

- Store the counter in an in-memory variable.
- Implement a GET `/visits/{page_id}` endpoint that returns the total number of visits.
    - The response of this endpoint should be:

{visits: [value], served_via: in_memory}

Testing Criteria:

- Ensure the counter increments on every API call.
- The `/visits/{page_id}` endpoint should return the correct count.

Task 2: Counter Using Redis (20 points)

Objective:

Modify the visit counter to store and retrieve visit counts from Redis instead of using an in-memory variable.

Requirements:

- Connect your backend API to Redis.
- Store the visit counter as a Redis key-value pair.
- Ensure that visit counts persist even after the application server restarts.
- The response of this endpoint should be:

{visits: [value], served_via: redis}

Testing Criteria:

- Restart the application server and ensure the counter does not reset.
- Verify that the Redis cache correctly updates and retrieves the count.

Task 3: Implement Application Layer Caching (20 points)

Objective:

Enhance performance by implementing an application-level caching layer that temporarily stores visit counts to reduce direct Redis **Read** queries and improve response times.

Requirements:

- Implement an additional caching layer within the application (can be an in memory map etc.).
- Set an expiration time (TTL) of 5 seconds on cached values.
- Fall back to Redis when the cache expires or when there's a cache miss.
- The response of this endpoint should be:

When served via server cache:

{visits: [value], served_via: in_memory}

When served via Redis Cache:

{visits: [value], served_via: redis}

Notes:

- All writes still go directly to Redis.
- Reads are served with application layer cache and if miss, the new value is fetched from Redis and cached.

Testing Criteria:

- Verify that the system reads from the cache instead of making redundant queries to Redis.
- Ensure the cache expires after the TTL and retrieves updated values from Redis.
- Simulate high-frequency requests and ensure performance improves with caching.

Task 4: Batching Write Requests to Redis (20 points)

Objective:

Optimize write performance by implementing a write batching mechanism that accumulates visits in memory and periodically flushes them to Redis, reducing the number of Redis operations.

Requirements:

- Store new visit counts in an in-memory buffer instead of writing directly to Redis.
- Implement a background process that flushes accumulated counts to Redis every 30 seconds.
- **When serving read requests, combine the persisted count from Redis with the pending counts in the memory buffer.**
- **When serving read requests from Redis (cache miss), flush out the buffer as well.**

Testing Criteria:

- Verify that new visits are correctly accumulated in the memory buffer.
- Ensure the background flush process successfully updates Redis every 30 seconds.
- Confirm that read requests return the correct total count (Redis + pending writes).cross shards in various scenarios, including shard failures or rebalancing operations.

Task 5: Implement Redis Sharding for Scalability (20 points)

Objective:

Ensure the visit counter can handle large-scale traffic by implementing Redis sharding, distributing data across multiple Redis instances to improve scalability and fault tolerance.

Requirements:

- Implement Redis sharding by partitioning visit count data across two Redis instances (put those instances at port 7070 and 7071).
- Use a consistent hashing mechanism to determine which shard stores which data.
- Implement logic in the backend API to query the appropriate Redis instance based on the user request.
- Ensure that sharding does not lead to data inconsistency or excessive inter-shard communication.
- Provide a mechanism for scaling the number of shards dynamically as traffic increases.

<u>When served via server cache:</u>

{visits: [value], served_via: in_memory}

<u>When served via Redis 7070:</u>

{visits: [value], served_via: redis_7070}

<u>When served via Redis 7071:</u>

{visits: [value], served_via: redis_7071}

Testing Criteria:

- Ensure visit counts are correctly distributed across Redis instances based on Consistent Hashing Strategy

- Validate that queries fetch the correct data from the appropriate shard.
- Simulate traffic spikes and ensure that the system scales effectively.
- Verify data consistency across shards in various scenarios, including shard failures or rebalancing operations.

Submission Guidelines

- Submit your code by filling this form: https://forms.gle/ckHMjRdqVf88h7YKA . The GitHub Repository should be a private repository with access to https://github.com/Naman-Bhalla/ and https://github.com/anshumansingh
- Include a README file with setup instructions.
- Provide API documentation for testing endpoints.
- Tech Stack to Use:
    - Python
    - FastAPI
    - Redis

Grading Rubric

Task	Points
Basic Visit Counter	20
Counter Using Redis	20
Implement Application Layer Caching	20
Batching Writes	20
Redis Sharding	20
Total	**100**

Happy Coding! 🚀