Scenario

You have a blog where each post has a view counter. Every time someone reads a post, the view count increases. Querying the database for every view is slow. Use Redis to make it faster!

Your Task (10 Points Total)

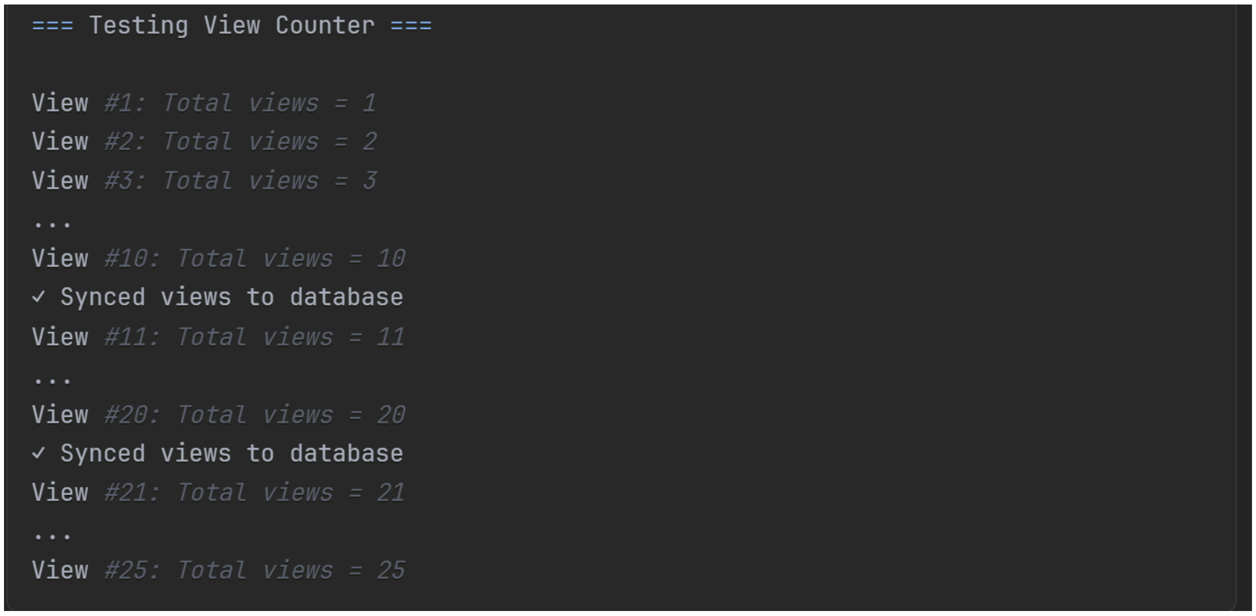
Using the demo code as a reference, add **ONE new method** to the DatabaseWithCache class:

Method to Implement: increment_post_views(post_id)

What it should do:

1. **Increment the view count in Redis** using the key "views:post:{post_id}"
   - Use Redis's INCR command: self.redis_client.incr(key)
2. **Every 10 views**, save the count back to the database
   - Check if view_count % 10 == 0
   - Update the database: UPDATE users SET views = ? WHERE id = ?
   - Print a message: "Synced views to database"
3. **Return the current view count**

Expected Output:



```
=== Testing View Counter ===

View #1: Total views = 1
View #2: Total views = 2
View #3: Total views = 3
...
View #10: Total views = 10
✓ Synced views to database
View #11: Total views = 11
...
View #20: Total views = 20
✓ Synced views to database
View #21: Total views = 21
...
View #25: Total views = 25
```

Task 2: Semantic Caching with Redis and Ollama

Assignment: Build a semantic caching system that stores Ollama LLM responses in Redis using vector embeddings. Configure Redis with a vector index to store query embeddings (using a model like SentenceTransformers), set a similarity threshold around 0.85, and implement logic that generates an embedding for each incoming query, searches Redis for semantically similar cached queries using cosine similarity, and returns the cached response if the similarity score exceeds your threshold. When no similar query exists, send the request to Ollama, get the response, then store both the query embedding and response in Redis for future cache hits.

You can reuse the llm calls from your last assignment if you want to.

IMPORTANT: Test your system with at least 10 diverse queries including exact duplicates, paraphrased questions, and completely new queries to evaluate performance.

Measuring Cache Performance: To determine whether a response came from cache or directly from Ollama, implement a **simple flag-based tracking system** in your query handler function.

When your similarity search finds a match above the threshold, set a boolean flag **`is_cached=True`** and log the cache hit along with the similarity score before returning the cached response. When no match is found and you call Ollama, set **`is_cached=False`** and log the cache miss.

Output Format:

Return this flag alongside your response so you can track metrics like cache hit rate (percentage of queries served from cache), average response time for cached vs non-cached queries, and the speed improvement factor. Add timestamps before and after each operation to measure that cached responses should be 4-15X faster than Ollama calls.

Add screenshots and code for the given tasks in your submission.