The document "System Design: The Sharded Counters" provides a comprehensive guide to understanding and implementing sharded counters in the context of system design for modern, high-traffic applications. Below is a detailed summary:

### **Problem Statement**

Real-time applications such as Facebook, Twitter, and YouTube experience high user traffic, resulting in multiple interactions (e.g., views, likes, comments) that occur rapidly. For instance, when an image is posted on a Facebook page with millions of followers, the number of likes can increase every millisecond. Managing the counting operation for such high-frequency interactions, especially when thousands of such images or videos are uploaded simultaneously by numerous celebrities with millions of followers, is challenging. This is referred to as the "heavy hitters problem."

### **Challenges**

The primary challenge lies in managing the performance and precision of counting operations under heavy load. The document cites examples like YouTube videos viewed by millions in 24 hours and Twitter handling 6,000 tweets per second (360,000 tweets per minute or about 500 million tweets per day). Handling billions of likes on these tweets each day requires a robust system to manage concurrent write operations efficiently. The key difficulty is that writing takes more time than reading, and as concurrent writes increase, the lock contention on a single counter also increases non-linearly, potentially leading to most of the time being spent acquiring locks for safe updates.

### **High-level Design of Sharded Counters**

1. **High-level Design**: The document introduces a high-level solution sketch for sharded counters. This includes a brief explanation of API design. The high-level design involves dividing the counter into multiple shards to distribute the load and reduce contention.
2. **Detailed Design**: The detailed design lesson delves deeply into the specifics of implementing sharded counters. This section also evaluates the proposed design, considering various factors such as load balancing, consistency, and fault tolerance.
3. **Quiz**: To reinforce the concepts covered, a quiz reviews the major ideas related to the design of sharded counters.

### **Detailed Design Considerations**

* **Sharding Strategy**: The counter is divided into multiple shards. Each shard maintains a part of the counter, reducing the load on any single node. When an update is required, it is directed to a specific shard.
* **Load Balancing**: Proper load balancing ensures that no single shard becomes a bottleneck. The distribution of write operations across multiple shards reduces the contention and increases throughput.
* **Consistency**: Ensuring consistency in a distributed system is critical. The design must handle scenarios where updates are happening concurrently across different shards.
* **Fault Tolerance**: The system should be resilient to failures. If a shard fails, mechanisms should be in place to recover or redistribute its load to maintain overall system performance.

### **API Design**

The API design for sharded counters includes methods for:

* **Incrementing the Counter**: This involves selecting the appropriate shard and updating the counter.
* **Reading the Counter**: This requires aggregating the counts from all shards to provide the total count.
* **Handling Failures**: Methods to deal with shard failures and ensure the system remains operational.

### **Conclusion**

The document concludes with a discussion on the importance of sharded counters in handling high-frequency interactions in modern applications. The provided design principles and detailed implementation strategies offer a robust solution to the heavy hitters' problem, ensuring scalable, consistent, and fault-tolerant performance.

This summary covers the key points from the document on sharded counters, offering a detailed look at the problem, high-level and detailed design strategies, and the importance of sharding in managing high-traffic real-time applications.

### **Summary: High-level Design of Sharded Counters**

#### **Problem Statement**

Real-time applications such as Facebook, Twitter, and YouTube face challenges in managing high user traffic. For instance, a single image posted on Facebook by a celebrity with millions of followers can receive likes at a rate of thousands per second. Similarly, YouTube videos can garner millions of views in a very short span, and Twitter handles thousands of tweets per second. Managing the counting operations for such high-frequency interactions becomes increasingly complex due to the need for precision and performance. This challenge is referred to as the "heavy hitters problem."

#### **Challenges**

The core challenge is handling the massive number of concurrent write operations. Writing data is inherently more time-consuming than reading it. As concurrent writes increase, lock contention on a single counter also increases non-linearly, leading to performance bottlenecks. Efficiently managing these high-volume writes and reads while maintaining data consistency and minimizing latency is crucial for a good user experience.

#### **High-level Solution Sketch**

To address these challenges, a high-level solution involves using sharded counters. Sharded counters distribute the load by dividing a counter into multiple shards, each running on different computational units. This distribution allows for parallel processing, reducing contention and improving performance.

##### **Conceptual Overview**

1. **Distributed Counters**: A sharded counter, or distributed counter, consists of multiple shards. Each shard is a smaller part of the overall counter.
2. **Load Distribution**: Write requests are distributed across these shards. When a write request (e.g., a like) is received, it is directed to a specific shard. This reduces the load on any single shard and enhances overall system performance.
3. **Reading from Shards**: When a read request is made, the system aggregates the values from all shards to provide the total count. This aggregation ensures that the system can handle high read loads efficiently.

#### **Detailed Example**

Consider a YouTube channel with millions of subscribers that uploads a new video. The server receives a large number of write requests for video views. A new counter is created for the video, and write requests are distributed across shards. Each shard's value is updated independently. When a read request is made to determine the total view count, the system aggregates the values from all shards.

#### **API Design for Sharded Counters**

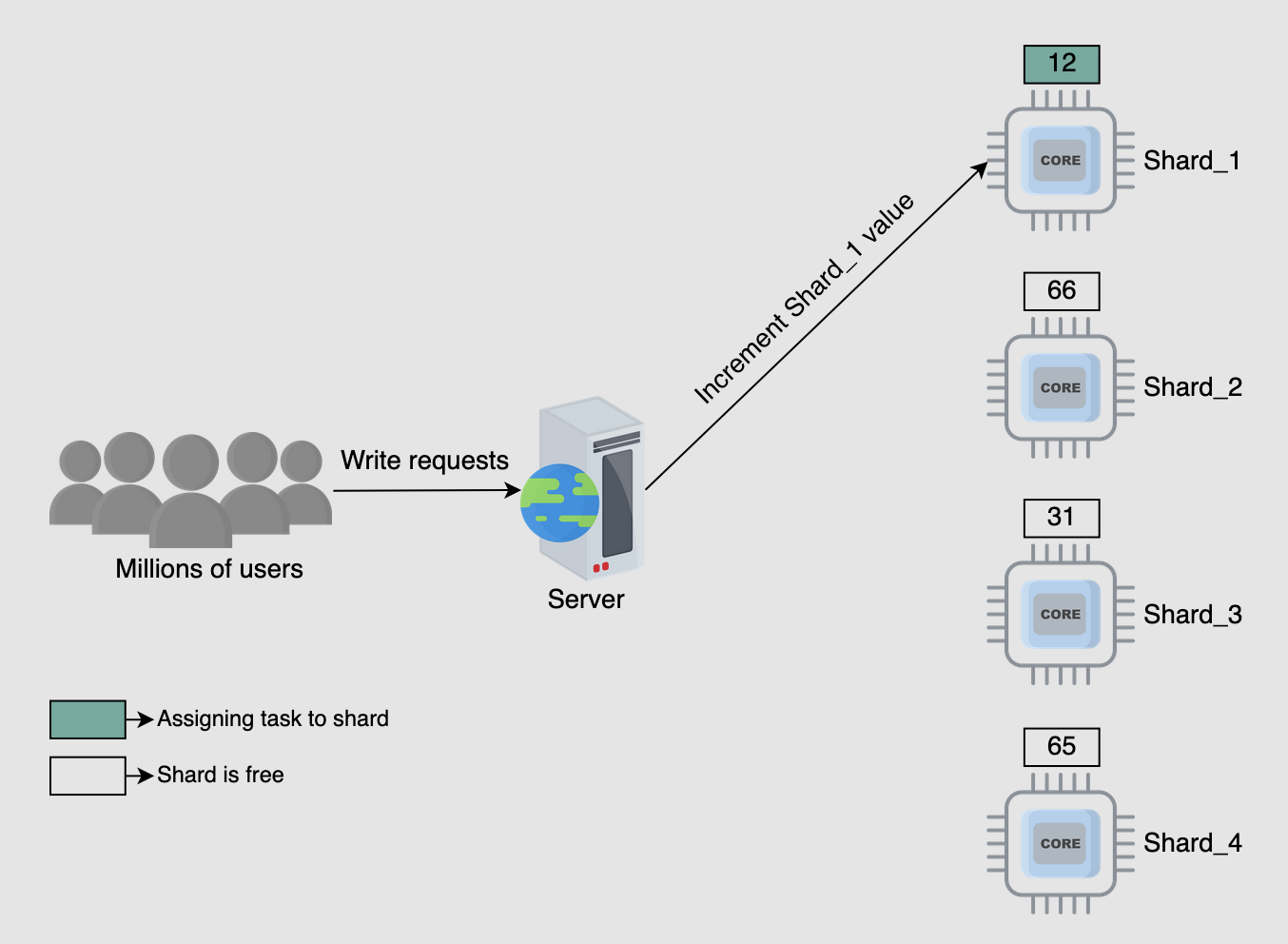
The API design for sharded counters includes functions for creating, writing to, and reading from counters. These APIs facilitate interactions between the sharded counters and their callers.

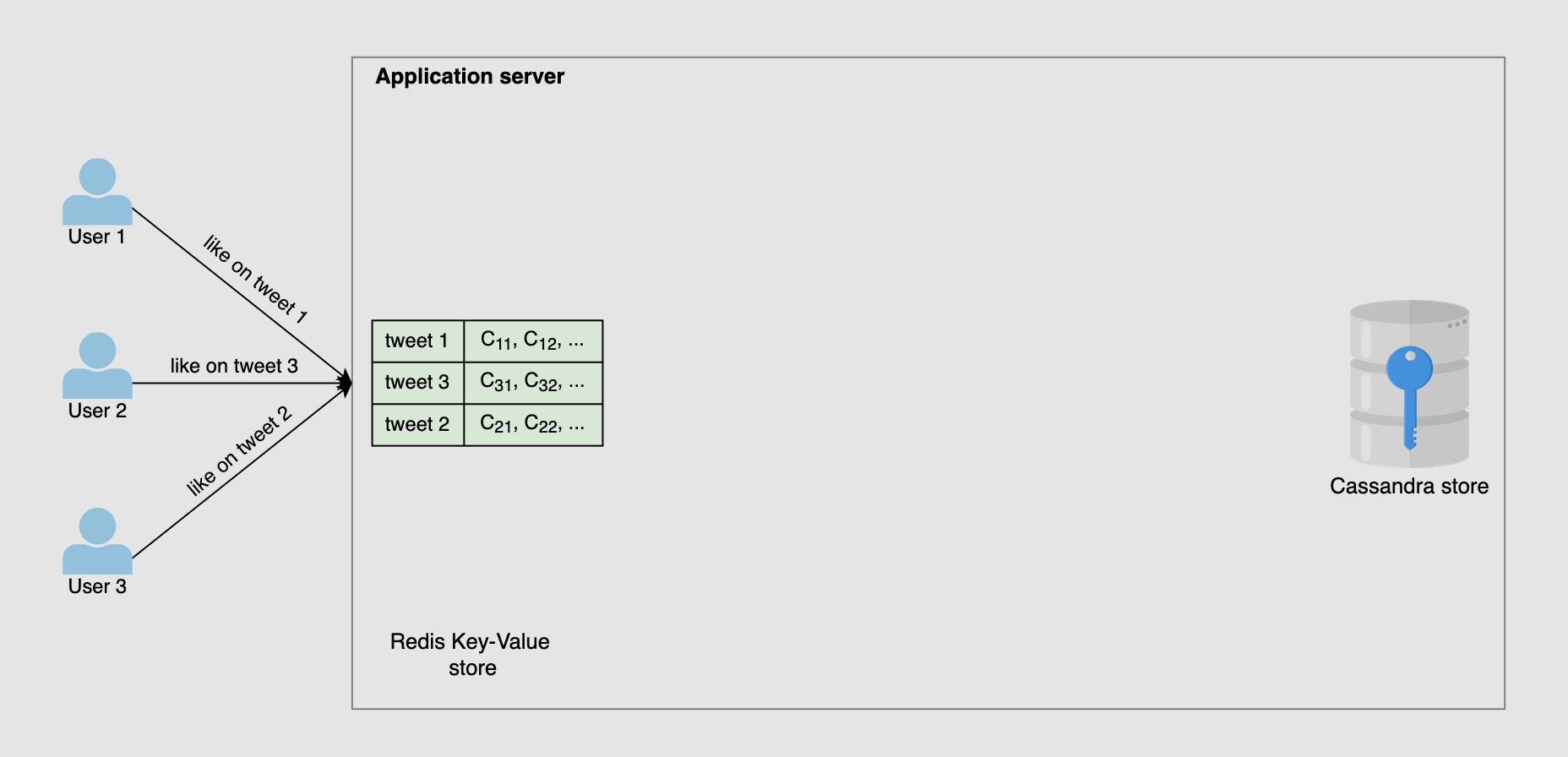
1. **Create Counter**: The createCounter API initializes a distributed counter with a specified number of shards.
   * createCounter(counter\_id, number\_of\_shards): counter\_id is a unique identifier, and number\_of\_shards specifies the number of shards.
   * The metadata for counters, including identifiers and shard mappings, is stored in an appropriate data store.
2. **Write Counter**: The writeCounter API handles incrementing or decrementing a counter by directing the operation to a specific shard.
   * writeCounter(counter\_id, action\_type): counter\_id is the unique identifier of the counter, and action\_type specifies whether to increment or decrement the counter.
   * This API is used for actions such as likes, replies, etc., on social media posts.
3. **Read Counter**: The readCounter API fetches the current value of the counter by aggregating values from all shards.
   * readCounter(counter\_id): counter\_id is the unique identifier of the counter.
   * For example, on Twitter, this API is used to fetch the number of likes or retweets on a tweet by aggregating the values from all associated shards.

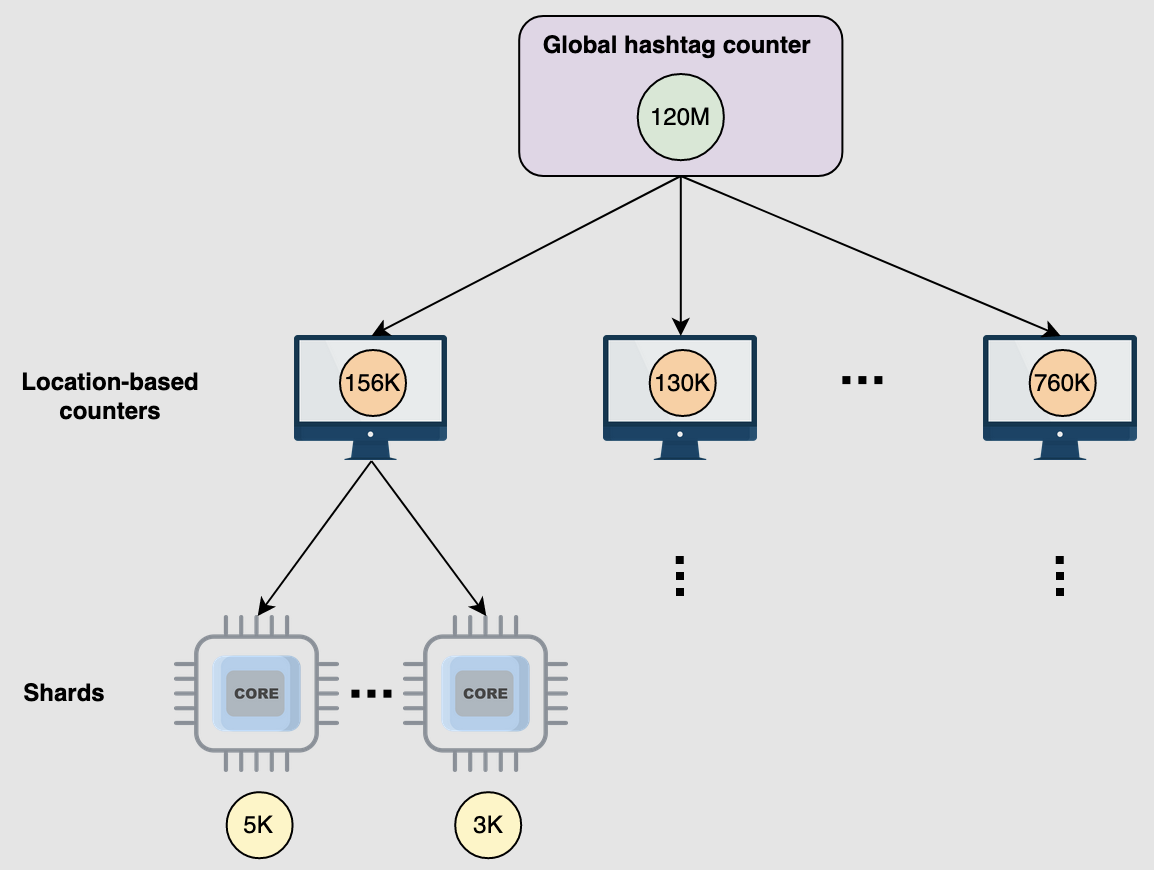
### **Conclusion**

Sharded counters provide a scalable and efficient solution to the heavy hitters problem in high-traffic real-time applications. By distributing load across multiple shards, they enhance performance and reduce contention, ensuring a smooth and responsive user experience. The API design supports easy integration and management of these distributed counters, making it a robust choice for handling high-frequency interactions in modern applications​

### **Detailed Design of Sharded Counters**







The document "Detailed Design of Sharded Counters" from the course "Grokking Modern System Design Interview for Engineers & Managers" provides an in-depth analysis and implementation strategy for sharded counters. Here's a detailed summary:

#### **Overview**

Sharded counters are used to handle high-frequency, high-volume write operations in real-time applications like social media platforms. These counters are essential for maintaining performance and ensuring scalability.

#### **Key Concepts**

1. **Sharded Counter Creation**:
   * When a user posts a tweet, multiple counters are created for each interaction type (like, reply, retweet, view).
   * The number of shards per counter depends on predicted write load. Factors like follower count and potential for virality influence this prediction.
   * The system must dynamically adjust the number of shards based on actual load to optimize performance.
2. **Handling Write Requests**:
   * **Round-robin Selection**: Requests are distributed sequentially across shards. This is simple but may not balance load efficiently.
   * **Random Selection**: Shards are selected randomly, which can also lead to load imbalance.
   * **Metrics-based Selection**: A load balancer selects shards based on their current load, providing a more balanced distribution.
3. **Handling Read Requests**:
   * Read requests aggregate values from all shards of a counter to provide the total count.
   * High write traffic can make it challenging to get real-time values, so periodic aggregation and caching are used to balance accuracy and performance.
4. **Using Sharded Counters for the Top K Problem**:
   * Sharded counters can solve the Top K problem by maintaining counts of hashtags or other metrics.
   * Counters are created for each hashtag, with shard counts adjusted based on usage.
   * Region-wise counters and time windows help manage trends and display relevant data to users.
5. **Placement of Sharded Counters**:
   * Shards can be placed on application servers, separate nodes in a data center, or at the network edge.
   * Placement depends on the use case and aims to optimize performance and handle heavy traffic efficiently.
6. **Evaluation of Sharded Counters**:
   * **Availability**: Multiple shards reduce the risk of a single point of failure.
   * **Scalability**: Shards allow horizontal scaling by adding more nodes to handle increased load.
   * **Reliability**: Sharded counters ensure that write requests are handled promptly, increasing system reliability.

#### **Conclusion**

Sharded counters are crucial for improving the performance of large-scale applications by providing high scalability, availability, and reliability. They address challenges like the heavy hitters problem and the Top K problem, making them indispensable in modern system design.

This detailed design covers the creation, management, and evaluation of sharded counters, providing a robust framework for handling high-volume write and read operations in real-time applications​

​ ​