**System Design: The Sharded Counters**

Get introduced to sharded counters.

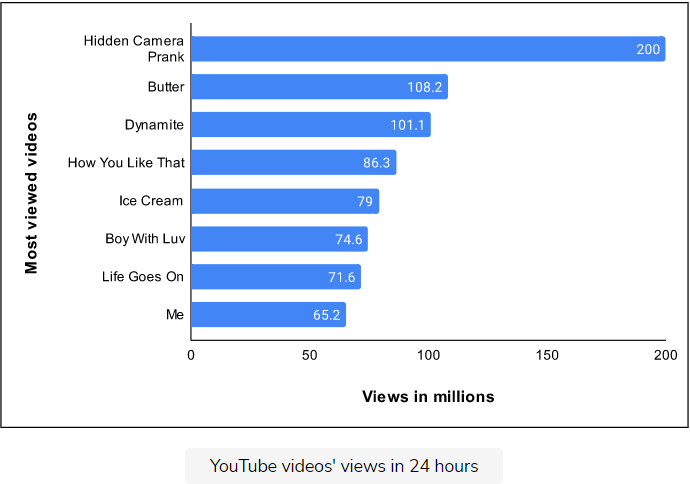
**We'll cover the following**

* [Problem statement](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/5881650651332608#Problem-statement)
* [How will we design sharded counters?](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/5881650651332608#How-will-we-design-sharded-counters?)

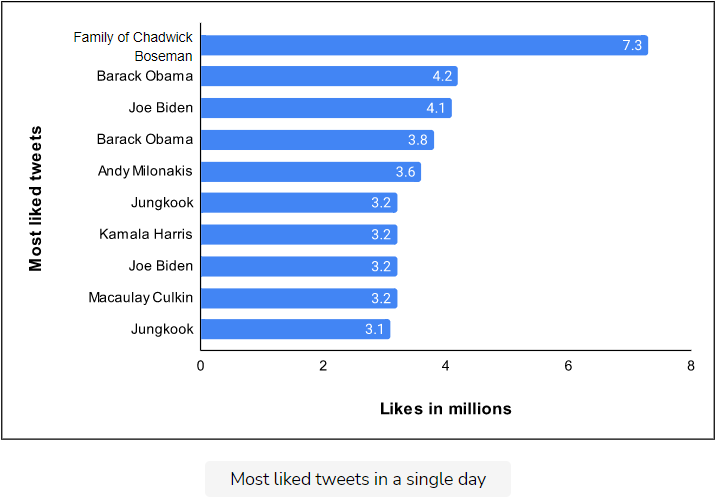
**Problem statement**

Real-time applications like Facebook, Twitter, and YouTube have high user traffic. Users interact with the applications and perform multiple operations (view, like, comment, and so on) depending on the application’s structure. For instance, an image is posted on a Facebook page that has millions of followers, and the post’s likes rapidly increase after each millisecond. Here, it might be easy to count the likes for this single image, but what will we do when thousands of such images or videos are uploaded simultaneously by many celebrities, each with millions of followers. This problem is known as the **heavy hitters problem**.

The above scenario shows how a simple counting operation becomes challenging to manage with precision and performance. The following figure shows YouTube’s videos that were viewed by millions of users in a 24-hour span in August 2021:



On average, six thousand tweets are sent on Twitter within one second, which equals 360,000 tweets per minute and about 500 million tweets per day. A challenging task is to handle billions of likes on these 500 million tweets per day. The following table shows the most liked tweets in one day as of 2022:



How will we handle millions of write requests coming against the likes on thousands of tweets per minute? The challenge is that writing takes more time than reading, and concurrent activity makes this problem harder. As the number of concurrent writes increases for some counter (which might be a variable residing in a node’s memory), the lock contention increases non-linearly. After some point, we might spend most of the time acquiring the lock so that we could safely update the counter.

**How will we design sharded counters?**

We have divided the design of sharded counters into three lessons:

1. [**High-level Design**](https://www.educative.io/collection/page/10370001/4941429335392256/6126387596886016): We’ll discuss the high-level design of sharded counters in this lesson. In addition, we’ll also briefly explain the API design.
2. [**Detailed Design**](https://www.educative.io/collection/page/10370001/4941429335392256/5076411916484608): This lesson will dive deeply into the design of sharded counters. Moreover, we’ll also evaluate our proposed design.
3. [**Quiz**](https://www.educative.io/collection/page/10370001/4941429335392256/5659634509021184): We’ll review major concepts of sharded counters design with a quiz.

Let’s begin with the high-level solution sketch of sharded counters.

**High-level Design of Sharded Counters**

Let's understand and design sharded counters.

**We'll cover the following**

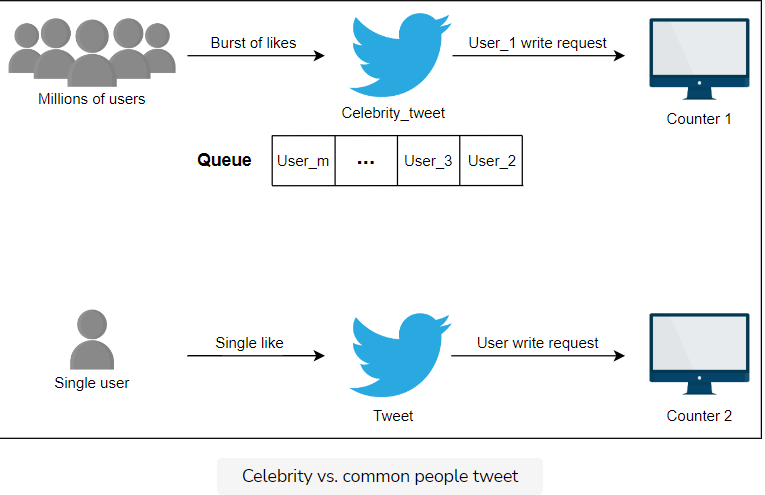
* [High-level solution sketch](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/6276216059592704#High-level-solution-sketch)
* [API design for sharded counters](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/6276216059592704#API-design-for-sharded-counters)
  + [Create counter](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/6276216059592704#Create-counter)
  + [Write counter](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/6276216059592704#Write-counter)
  + [Read counter](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4716688570580992/6276216059592704#Read-counter)

**High-level solution sketch**

Managing millions of tweet likes requires many counters operating on many nodes. To manage these counters, we need an efficient system that can provide high performance and scalability as the number of users grows.

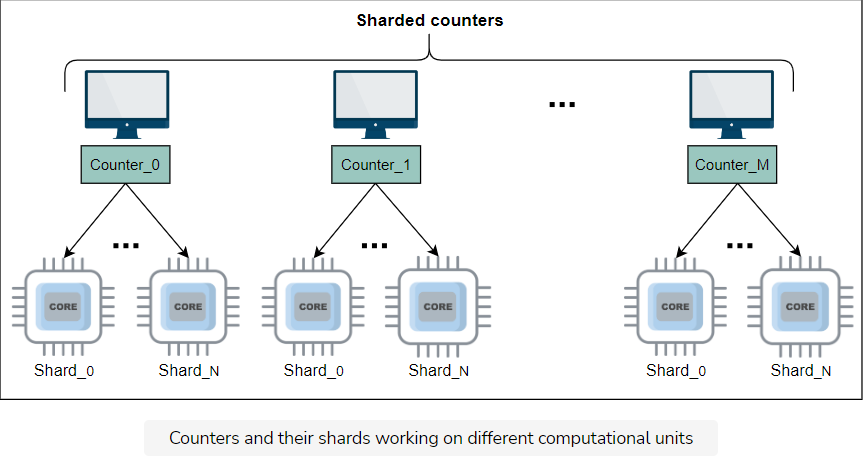
What will happen when a single tweet on Twitter gets a million likes, and the application server receives a write request against each like to increment the relevant counter? These millions of requests are eventually serialized in a queue for data consistency. Such serialization is one way to deal with concurrent activity, though at the expense of added delay. Real-time applications want to keep the quality of experience high by providing as minimum as possible latency for the end user.

Let’s see the illustration below to understand this problem:



A single counter for each tweet posted by a celebrity is not enough to handle millions of users. The solution to this problem is a **sharded counter**, also known as a distributed counter, where each counter has a specified number of shards as needed. These shards run on different computational units in parallel. We can improve performance and reduce contention by balancing the millions of write requests across shards.

First, a write request is forwarded to the specified tweet counter when the user likes that tweet. Then, the system chooses an available shard of the specified tweet counter to increment the like count. Let’s look at the illustration below to understand sharded counters having specified shards:



In the above illustration, the total number of shards per counter is (N+1)(*N*+1). We’ll use an appropriate value for N*N* according to our needs. Let’s discuss an example to understand how sharded counters handle millions of write and read requests for a single post.

Let’s assume that a famous YouTube channel with millions of subscribers uploads a new video. The server receives a burst of write requests for video views from worldwide users. First, a new counter initiates for a newly uploaded video. The server forwards the request to the corresponding counter, and our system chooses the shard randomly and updates the shard value, which is initially zero. In contrast, when the server receives read requests, it adds the values of all the shards of a counter to get the current total.

We can use a sharded counter for every scenario where we need scalable counting (such as Facebook posts and YouTube videos).

## API design for sharded counters

This section discusses the APIs that will be called for sharded counters. Our API design will help us understand the interactions between sharded counters and their callers. To make our discussion concrete, we’ll discuss each API function in the context of Twitter. Let’s develop APIs for each of the following functionalities:

* Create counter
* Write counter
* Read counter

Although the above list of API functions is not exhaustive, they represent some of the most important ones.

### Create counter

The \createCounter API initializes a distributed counter for use. The \createCounter API is given below:

createCounter(counter\_id, number\_of\_shards)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| counter\_id | It represents the unique ID of the counter. The caller of this API can use a sequencer to get a unique identifier. See the lesson on sequencer building blocks for more details. |
| number\_of\_shards | It specifies the number of shards for the counter. |

We can use an appropriate data store to keep our metadata, which includes counter identifiers, their number of shards, and the mapping of shards to physical machines.

Let’s consider Twitter as an example to understand how an application uses the above API. The \createCounter API is used when a user posts something on social media. For instance, if a user posts a tweet on Twitter, the application server calls the \createCounter API. The content\_type parameter is the post type that the system uses to decide the number of counters that need to be created. For example, the system needs a view counter only if the tweet contains a video clip.

To find an appropriate value for number\_of\_shards, we can use the following heuristics:

* The followers\_count parameter denotes the followers’ count of the user who posts a tweet.
* The post\_type parameter specifies whether the post is public or protected. Protected tweets are for the followers only, and in this case, we have a better predictor of the number of shards.

### Write counter

The \writeCounter API is used when we want to increment (or decrement) a counter. In reality, a specific shard of the counter is incremented or decremented, and our service makes that decision based on multiple factors, which we’ll discuss later. The \writeCounter API is given below:

writeCounter(counter\_id, action\_type)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| counter\_id | It is the unique identifier (provided at the time of counter creation). |
| action\_type | It specifies the intended action (increment or decrement value of the counter). We extract the required information about the counter from our data store. |

In our Twitter example, the \writeCounter API is used when users act (by liking, replying, and so on) on someone else’s post or their own post.

### Read counter

The \readCounter API is used when we want to know the current value of the counter. Our system fetches appropriate information from the datastore to collect value from all shards. The \readCounter API is given below:

readCounter(counter\_id)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| counter\_id | It is the unique identifier (provided at the time of counter creation).  For Twitter, the counter\_id will be decided based on the following metrics:  The tweet\_id specifies the tweet's unique ID for which the request is generated. We can use tweet\_id to get the counter\_id for all the counters of the features (likes, retweets, and so on). |

The \readCounter API is called when users want to see the number of likes or view counts on a specific tweet. Usually, this API is triggered by another API when users want to see their home or user timeline.

The following section will discuss what happens in the back-end system when all the above APIs are called.