High-level Design of Sharded Counters

Let's understand and design sharded counters.

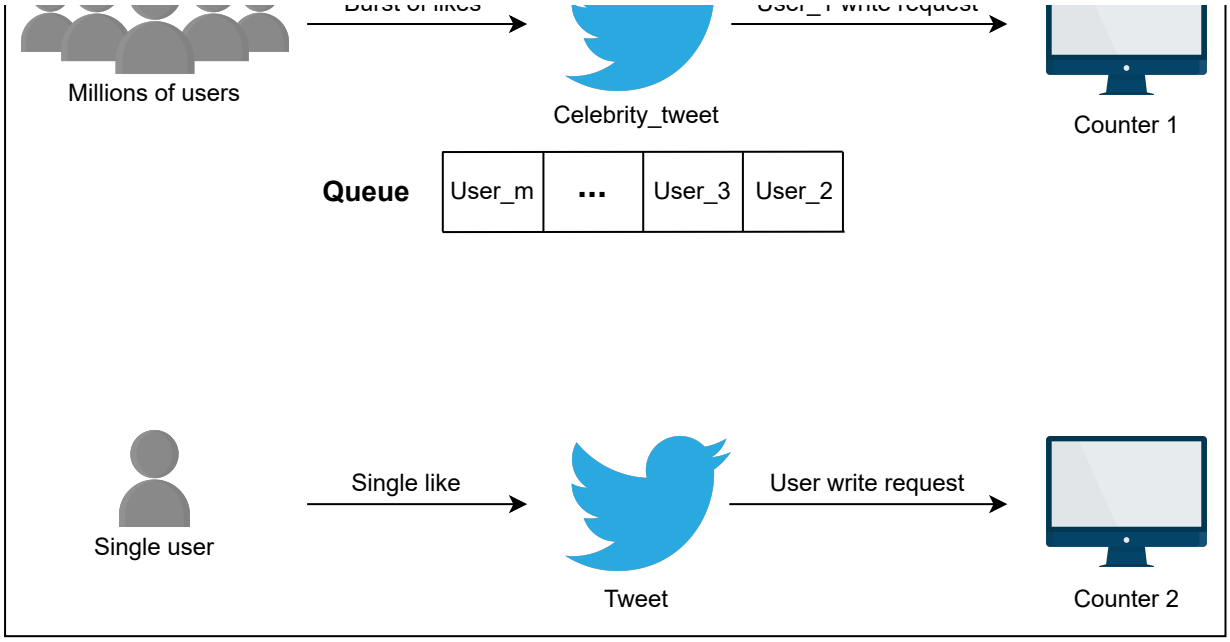
We'll cover the following

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.

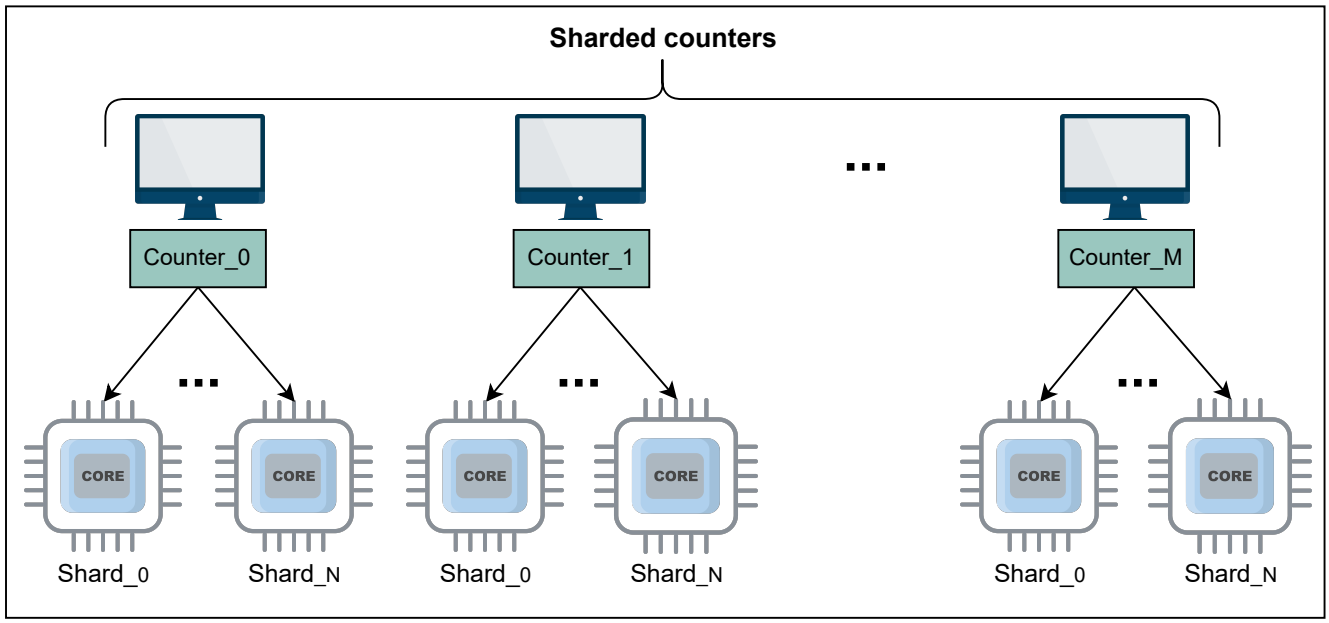
Let’s see the illustration below to understand this problem:



Celebrity vs. common people tweet

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:



Counters and their shards working on different computational units

In the above illustration, the total number of shards per counter is $(N + 1)$. We'll use an appropriate value for 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.