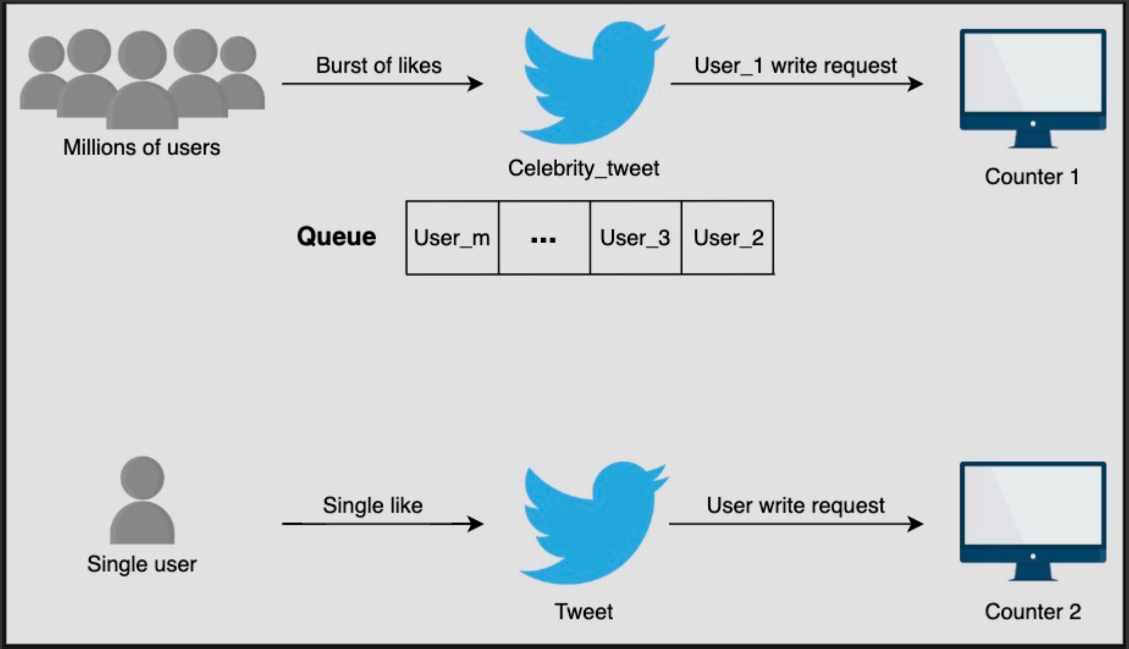
On average, six thousand tweets are sent on Twitter within one second, which equals 360,000 tweets per minute and about 500 million tweets per day. A challenging task is to handle billions of likes on these 500 million tweets per day. The following table shows the most liked tweets in one day as of 2022:

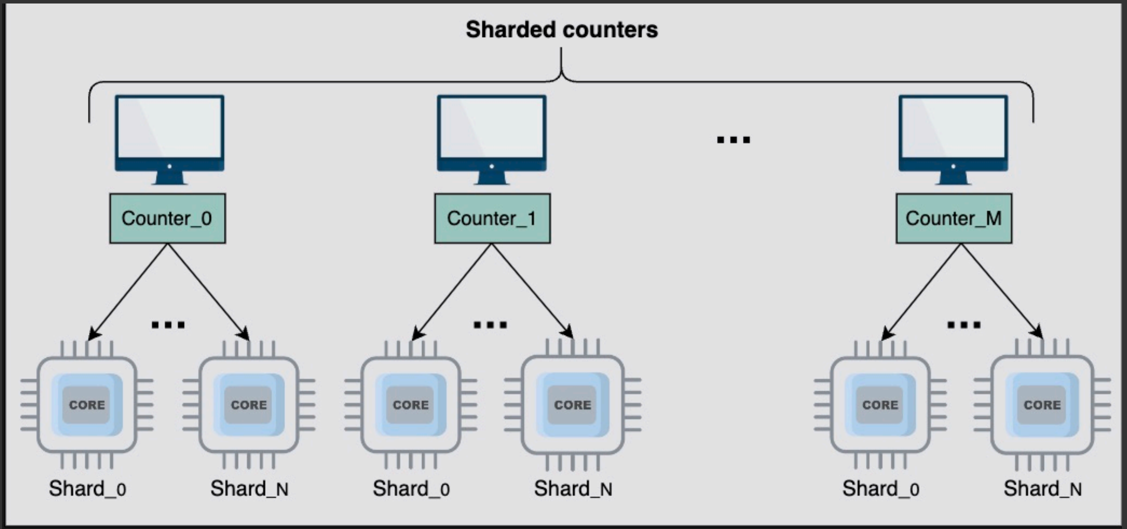
How will we handle millions of write requests coming against the likes on thousands of tweets per minute? The challenge is that writing takes more time than reading, and concurrent activity makes this problem harder. As the number of concurrent writes increases for some counter (which might be a variable 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.

High - level Design



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.



Counters and their shards working on different computational units

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.

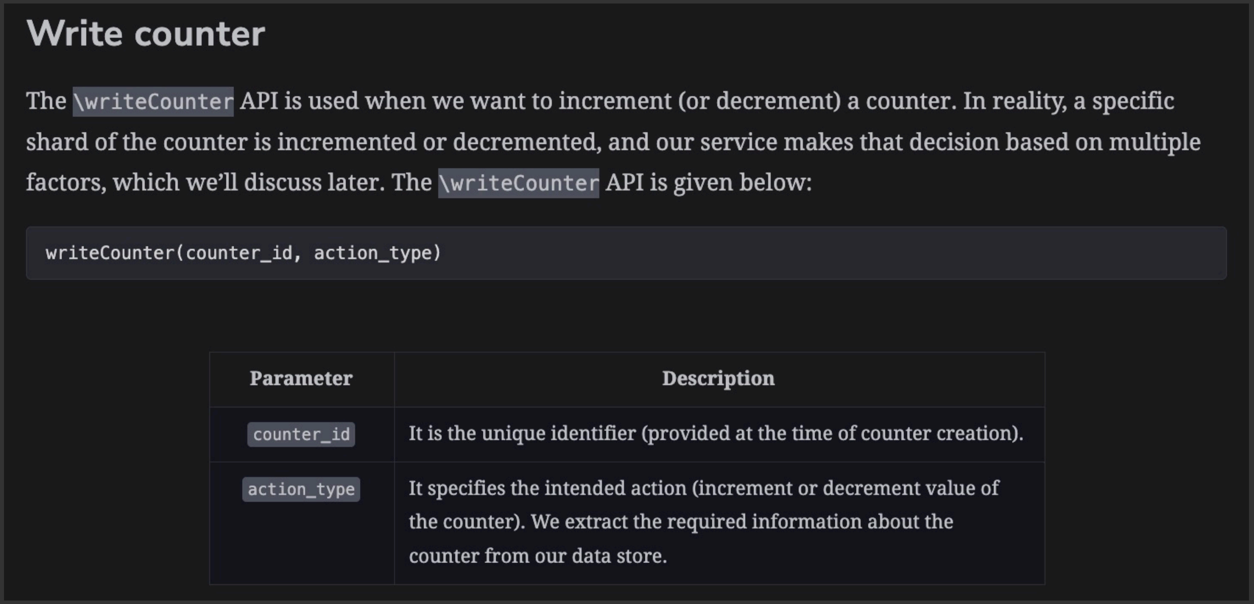
API design for Sharded Counters

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.

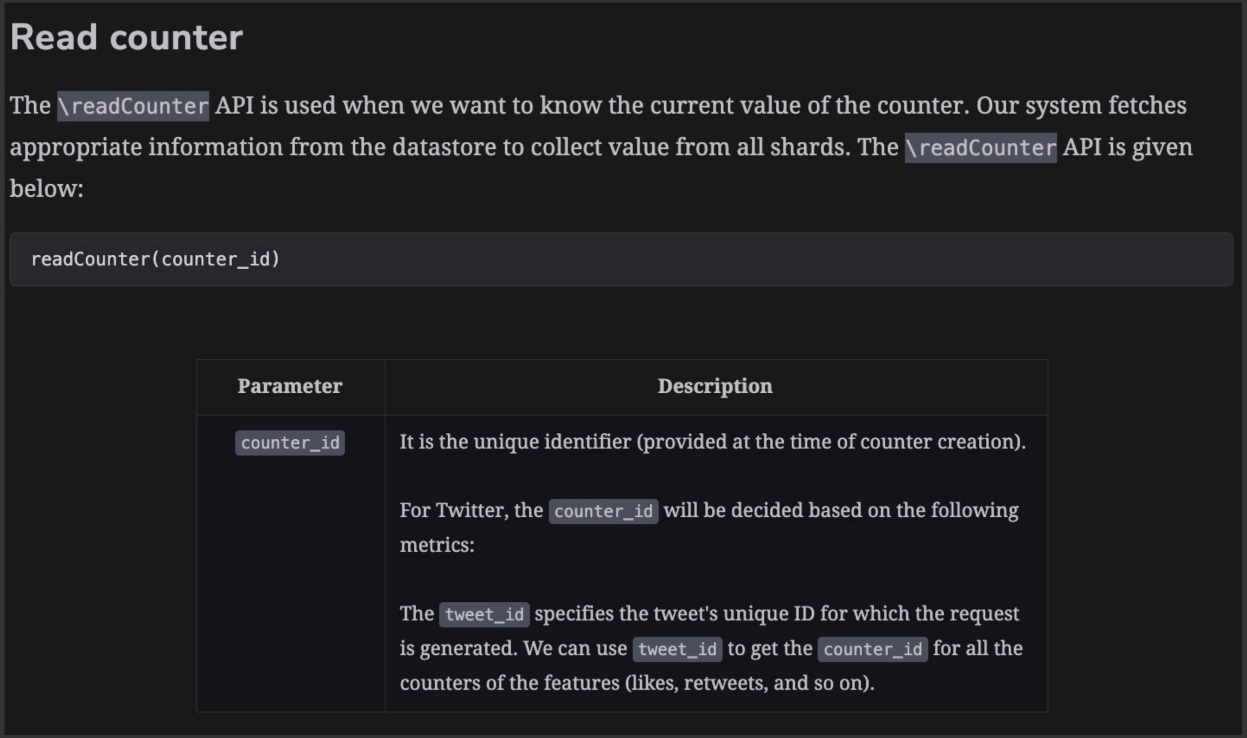


Write counter

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.



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).

Detailed design of Sharded Counters

How many shards should be created against each new tweet?
How will the shard value be incremented for a specific tweet?
What will happen in the system when read requests come from the end users?

the question is how does the system decide the number of shards in each counter? The number of shards is critical for good performance. If the shard count is small for a specific write workload, we face high write contention, which results in slow writes. On the other hand, if the shard count is too high for a particular write profile, we encounter a higher overhead on the read operation.

The decision about the number of shards depends on many factors that collectively try to predict the write load on a specific counter in the short term. For tweets, these factors include follower count. The tweet of a user with millions of followers gets more shards than a user with few followers on Twitter because there is a possibility that their tweets will get many, often millions, of likes.

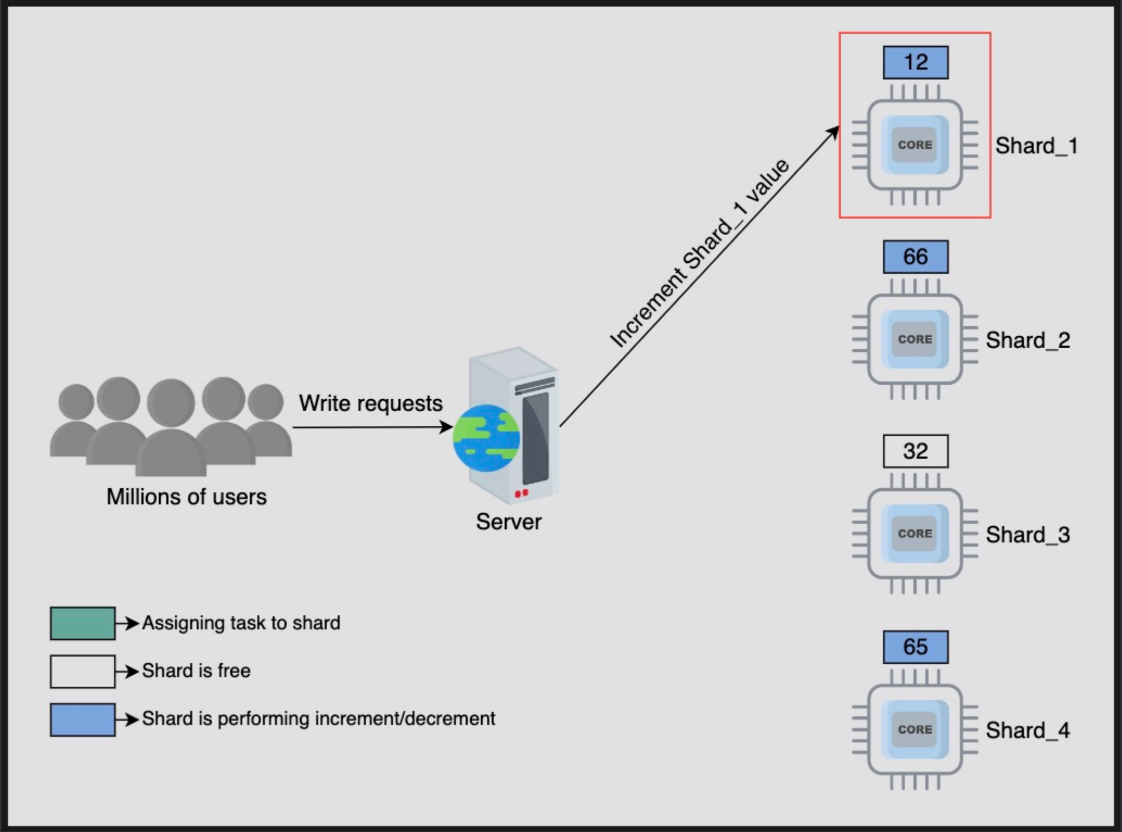
What happens when a user with just a few followers has a post go viral on Twitter?

The system needs to detect such cases where a counter unexpectedly starts getting very high write traffic. We'll dynamically increase the number of shards of the affected counter to mitigate the situation.

How does the system select these shards operating on different computational units (nodes) to assign the write requests?

Round-robin selection

One way to solve the above problem is to use a round-robin selection of shards. For example, let’s assume the number of shards is 100. The system starts with shard_1, then shard_2, and continues until it reaches shard_100. Usually, round-robin work allocation either overloads or underutilizes resources because scheduling is done without considering the current load conditions. However, if each request is similar (and roughly needs the same time to serve), a round-robin approach can be used and is attractive for its simplicity.

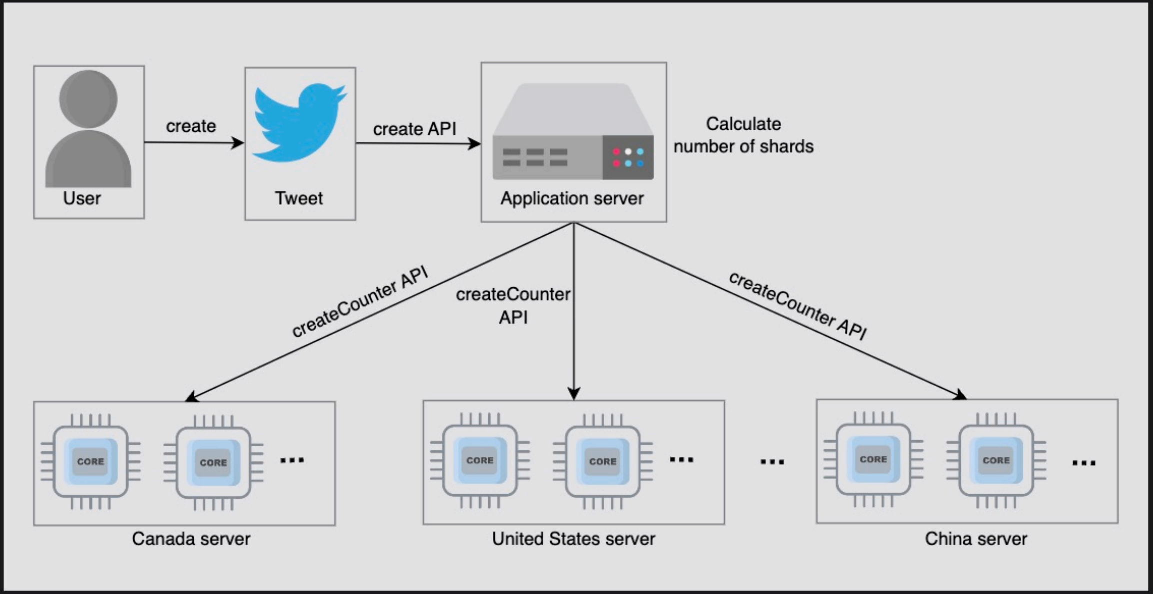


Random selection

Another simple approach can be to uniformly and randomly select a shard for writing. The challenge with both round-robin selection and random selection is with variable load changes on the nodes (where shards are hosted). It is hard to appropriately distribute the load on available shards. Load variability on nodes is common because a physical node is often being used for multiple purposes.

Metrics-based selection

The third approach is shard selection based on specific metrics. For example, a dedicated node (load balancer) manages the selection of the shards by reading the shards' status. The below slides go over how sharded counters are created:



Requests to create shards for a specified counter using createCounter API are dispatched to geographically dispersed servers

Manage read requests
When the user sends the read request, the system will aggregate the value of all shards of the specified counter to return the total count of the feature (such as like or reply). Accumulating values from all the shards on each read request will result in low read throughput and high read latency.

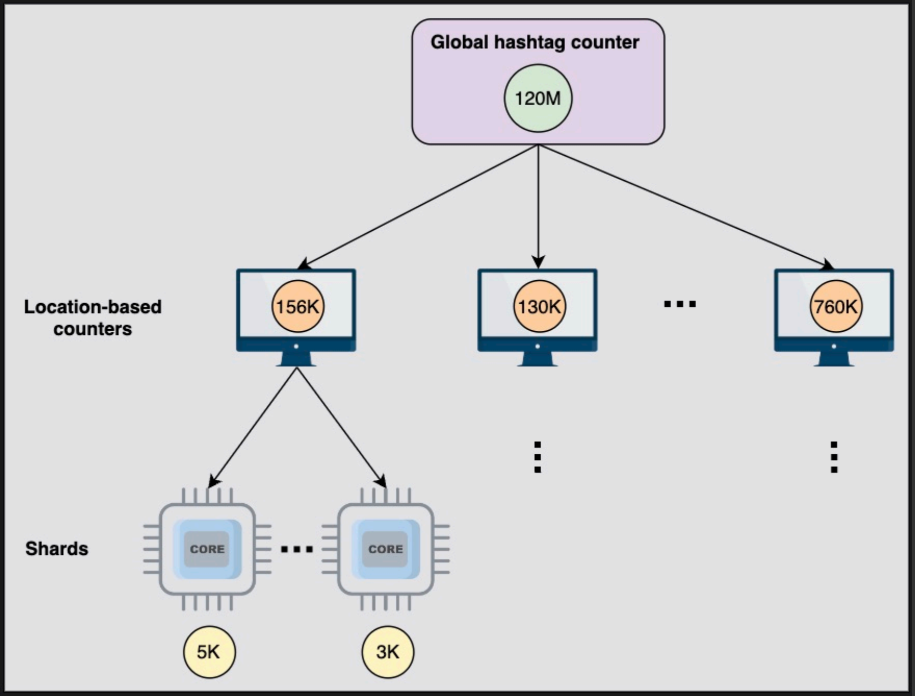
The decision of when the system will sum all shards values is also very critical. If there is high write traffic along with reads, it might be virtually impossible to get a real current value because by the time we report a read value to the client, it will have already changed.

Can you think of a use case where sharded counters with the above-mentioned consistency model might not be suitable?

We might not use shared counters with a relaxed consistency model where we need strong consistency. An example can be read-then-write scenarios where we first need to get the accurate value of something before deciding to modify it (actually, such a scenario will need transaction support).

Using sharded counters for the Top K problem

This section will discuss how we can use sharded counters to solve a real-world problem known as the **Top K** problem. We'll continue to use the real-time application Twitter as an example, where calculating trends is one of the Top K problems for Twitter. Here, K represents the number of top trends. Many users use various hashtags in their tweets. It is a huge challenge to manage millions of hashtags' counts to show them in individual users' trends timelines based on their locality.



The system calculates the 24-hour count of the specified hashtag

The global hashtag counter represents the total of all location-based counters.

Location-based counters represent their current count when the system reaches the set threshold in a specified time and the hashtag becomes a trend for some users. For example, Twitter sets 10,000 as a threshold. When location-based hashtag counts reach 10,000, Twitter shows these hashtags in the trends timeline of the users of the respective country where the hashtag is being used. The specified hashtag may be displayed worldwide if counts increase in all countries.

Should we lock all shards of a counter before accumulating their values?

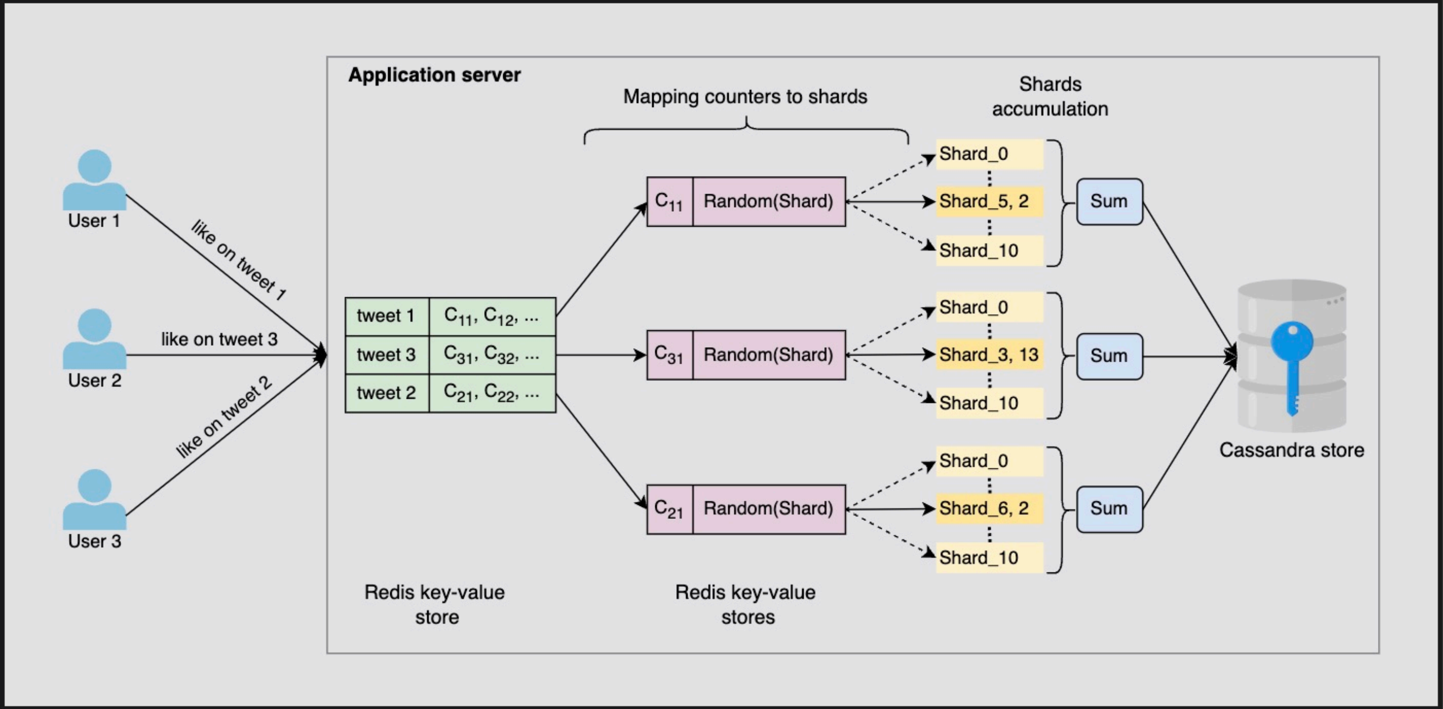
Hide Answer

No. Reads can happen concurrently with writes without the need for an across-shards lock. This lock will decimate the write performance, the original reason we used sharded counters. Under a relaxed consistency model, where the value of a counter might not reflect the exact current value, there is no need for simultaneous read locks across all shards.

However, depending on the specific use case, such a mechanism might be used when reads frequency is very low.

Reads can store counter values in appropriate data stores and rely on the respective data stores for read scalability. The Cassandra store can be used to maintain views, likes, comments, and many more counts of the users in the specified region. These counts represent the last computed sum of all shards of a particular counter.

When users generate a timeline, read requests are forwarded to the nearest servers, and then the persisted values in the store can be used to respond. This storage also helps to show the region-wise Top K trends.



All the computed sums are stored in the Cassandra store

Evaluation of the sharded counters

This section will evaluate the sharded counters and explain how sharded counters will increase performance by providing high availability and scalability.

Availability

A single counter for any feature (such as like, view, or reply) has a high risk of a single point of failure. Sharded counters eliminate a single point of failure by running many shards for a particular counter. The system remains available even if some shards go down or suffer a fault. This way, sharded counters provide high availability.

Scalability

Sharded counters allow high horizontal scaling as needed. Shards running on additional nodes can be easily added to the system to scale up our operation capacity. Eventually, these additional shards also increase the system's performance.

Reliability

Another primary purpose of the sharded counters is to reduce the massive write request by mapping each write request to a particular shard. Each write request is handled when it comes, and there is no request waiting in the queue. Due to this, the hit ratio increases, and the system's reliability also increases. Furthermore, the system periodically saves the computed counts in stable storage—Cassandra, in this case.

Conclusion

Nowadays, sharded counters are a key player in improving the overall performance of giant services. They provide high scalability, availability, and reliability. Sharded counters solved significant issues, including the heavy hitters and Top K problems, that are very common in large-scale applications.