Trending topics for Twitter

1. Functional requirements (5 minutes).

Design “Trending topics” functionality for Twitter.
System should be able to ingest hashtags.
What is the time span for trending topics (1 day, 1 week, etc)? 24 hours
Do we want real-time trending topics? (streaming or batching)?
Is it just by frequency of occurrence, or are there other things like influencers etc?

Use cases:

- User views a page with a hashtag.
- User navigates to a “Trending topics” page where hashtags with the most view during last 24 hours can be found

Non-requirements:

- Frontend service
- Searching and displaying topics by hashtag
- Storing viewing the pages with hashtags
- Location specific trending topics

Consistency vs availability - service must be **available** [Weak/strong/eventual => consistency | Failover/replication => availability]
Latency and Throughput requirements

2. Estimations. Back of the envelope calculations

- Twitter has 145M daily active users.
- Users post 500M tweets per day on average (350K writes per minute or 6K writes per second and **peak load is 12K QPS per second**)
- Let’s say we have 2x more reads than writes
- Peak load can be 10 * 12K = **120K reads per second**
- We have 120K read events per second to digest by our system.

3. Bucketsize functional requirements to microservices (5 minutes)

API Gateway. Load balancing. SSL termination. Rate limiter. Receives requests from users and depending on geographical location and load redirects the request to stateless frontend service.

Post service. The service allows users to save and view topics. It uses a cache to get the most recently viewed posts. When the service receives the request to view a topic, we can log this

event to the message queue either async or we can use a log and publish this event using an external service running on the same machine.

Message queue. Receives and delivers information about page views. The reason we want to use the message queue is that we don't want to affect performance of the topics service with a request that is not needed to get a response with post information.

Streaming processing. We can use Storm and implement a sliding window algorithm to compute the topics in real time. It aggregates and persists the top trending topics for the last 24 hours. It gets a hashId and a timestamp from the message queue, aggregates them and puts the aggregated trending topics to the database every N minutes (let's say 5 minutes).

Trending topics database. The database keeps trending topics in the format like:

Timestamp	**Sorted List (HashtagId - Count)**

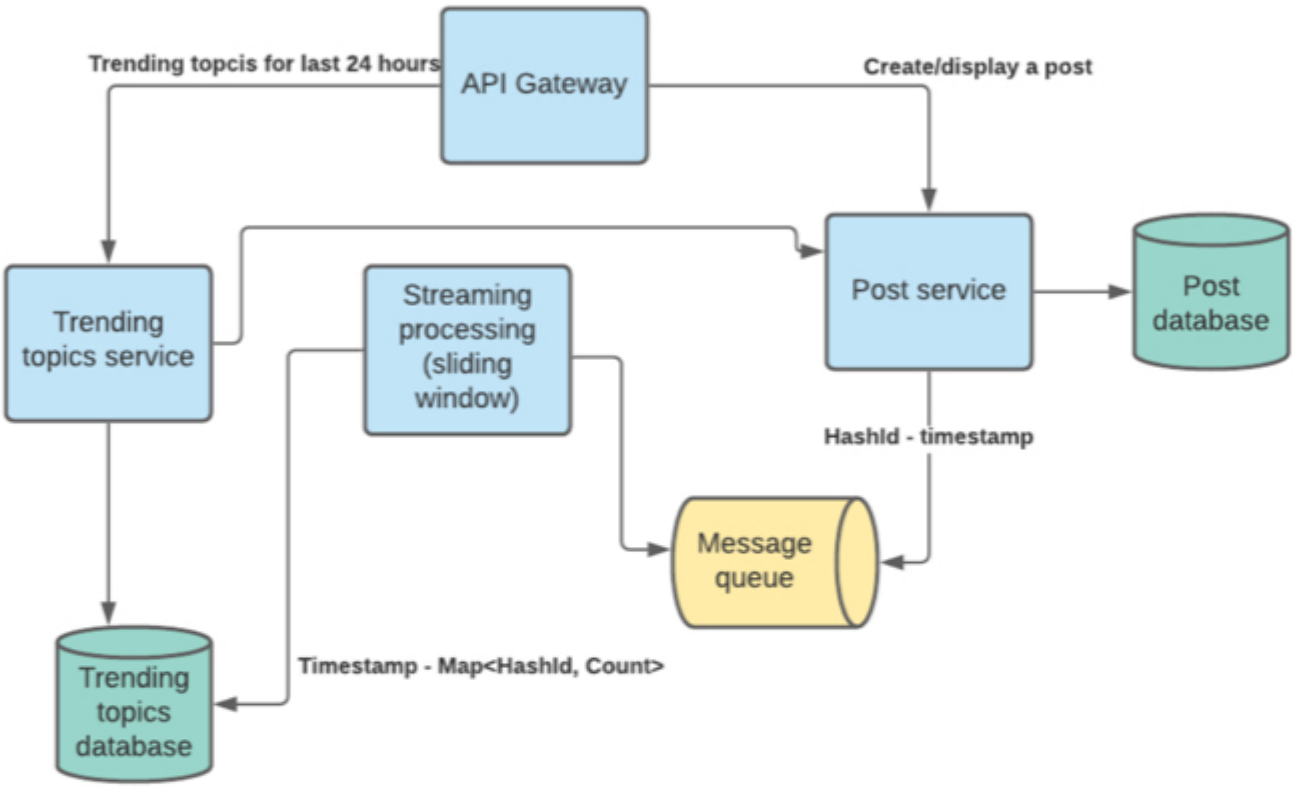
Trending topics service. Receives request:

```
GetTrendingTopics:
        Timestamp startTime;
        int periodSeconds;
        int count;
```

It loads all data for this period from the database, aggregates it using a priority queue and returns a response with the list of hashtags and number of views.

Rest API may look like:
GET /v1/trending?count=100&periodSeconds=3600

3. Draw the logical architecture. High level design (5-10 minutes)



We keep the following key-value in the Post database:
key:hashTagId - value:hashTagName
In the trending streaming processor we use hashTagId to save some space. Later Trending topics service extract hashtag names from Post service by hashTagId.

4a. Deep dive to Trending service and streaming processing (15 minutes).

This is a depth oriented problem.

First thing to clarify is whether we need trending topics for the last 24 hours (or whatever is reasonable time) or for all time using a formula to promote recent topics.

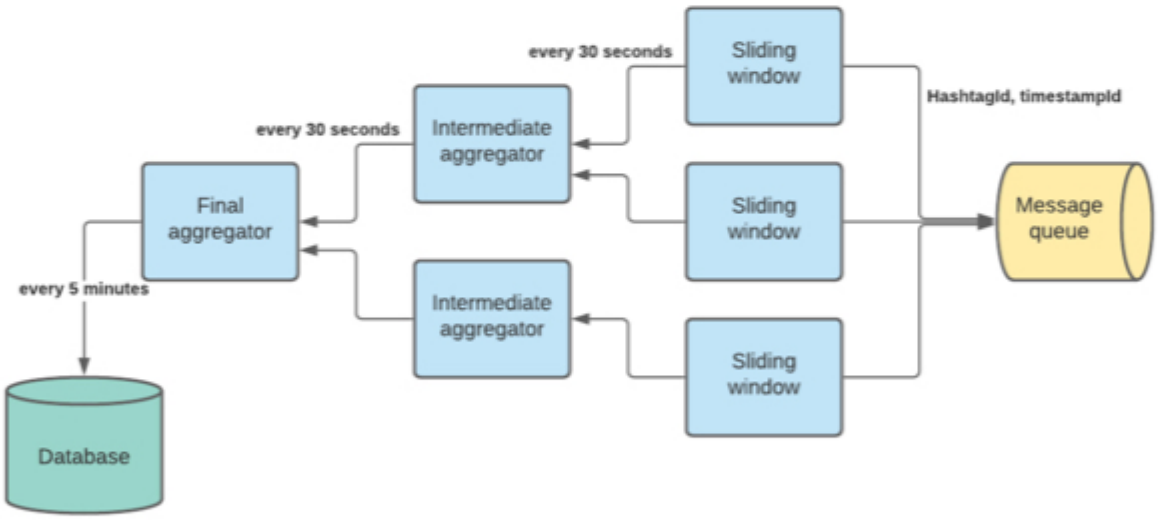
Advanced (unexpected).

For the latter we can use a z-score but most interviewers will not expect you to come up with a smart score rating.
If we want to calculate z score for each topic and find the topic with the largest z score.
z-score = ([current trend] - [average historic trends]) / [standard deviation of historic trends]

The **standard deviation** is a measure of the amount of variation or dispersion of a set of values.
Difference between current and average tells us nothing about how big the deviation from average.
That's why we divide the difference on standard deviation.
When a z-score is used, the higher or lower the z-score the more abnormal the trend, so for example if the z-score is highly positive then the trend is abnormally rising, while if it is highly negative it is abnormally falling. So once you calculate the z-score for all the candidate trends the highest 10 z-scores will relate to the most abnormally increasing z-scores.

Let's say we want to see trending topics only for the last 24 hours and we also want to keep the history of trending topics.

Let's take a look at our streaming processing service. If we want to have the latest 100 trending topics, we may use a topology like this:



Sliding window processor reads hashtags and timestamps from message queue and keeps a hashmap like:
topicId -> array of timeslots

Streaming processing.
The size of the array is the number of seconds after which we emit data. For example:

	Second 1	Second 2	Second 3	...	Second 30
#microservices	5	3	8		11

We rotate this sliding window to avoid creating/deleting new objects or changing the size of the array.
Every 30 seconds we emit an aggregated count for each topic to an **intermediate aggregator**. We should use shuffling by topic (similar to map/reduce shuffling) so intermediate aggregators always receive information about the same topics. Because intermediate aggregators have all counts for the topic, they can use priority queue to keep only the top 100 topics. Every 30 seconds intermediate aggregators send information about 100 topics to the final aggregator that aggregates top 100 topics across all intermediate aggregators. It also persists the result in the database. There is only 1 final aggregator.

Timestamp	**Sorted List (HashtagId - Count)**

Trending topics service.
It receives the request from API Gateway to get top N (up to 100) trending topics for a given time period. It creates a range query for the given time period and aggregates the result. We can keep the result in cache for several seconds since the query can be expensive and it's okay to have a little stale data.

4b. Gather non functional / capacity requirements and check whether and how to scale each tier.

120K events per second. Let's say we may have 10K unique hashtags per day.
Let's consider the Sliding window aggregator. It updates the counter in the hashmap and emits aggregated values every 30 seconds.

4c. Check if each tier needs to horizontally scale.

Sliding windows aggregator.

Memory
Memory needed to create a sliding window for 10K unique hashtags (we don't keep old hashtags and remove them when we emit aggregated counters):
hashTagId - long (8 bytes)
Counter - integer (4 bytes), for 30 seconds we need 30 counters.
10K * 8 + 10K * 30 * 4 = 80K + 1.2Mb. We may also have hashmap and array overhead so it can be like 2Mb. It's a pretty small number and the service will not be memory bound.

Throughput.
Input data: a server has 200Gb of RAM, 10 cores, 4Tb storage.
One event size is 8 bytes (timestamp) + 8 bytes (hashtag id) = 16 bytes.
TCP/IP header is 120 bytes.
Sliding window aggregators process all data in memory. They read events from the message queue and increment counters. We have 10 cores so we can process 10 events in parallel.

Processing 1 event should take less than 1ms (lock/unlock 50 ns? + main memory reference ~6*100 ns + read 1k from main memory 250 ns = 1000ns = 1us). Instead of reading a single event at a time from the queue, we can read several events and batch process them:
100 * 1us = 100 us

Reading from the network will take 20 us (1Kb over 1Gb network takes 10 us):
100 * 16 (100 view events) + 120(header) = 1720 ~=2K

Total time to process 100 view events is 120 us. Single thread can process ~=800 events in 1ms and 800K events in 1 second.

Sending counters to intermediate aggregator:
Size = 1k hashtags * 8 bytes + 1k counters * 4 bytes = 12K = 120us.

In real life there is also some additional overhead for garbage collection (if we use java), creating new sliding windows for new hashtags. Given that let's assume a single node can digest 100K events per second. Since we need redundancy, we can use 2 or 3 machines for sliding window aggregators.

Intermediate aggregator.

Memory
We keep a priority queue in memory that contains only the top 1000 hashtags and their counter.
1K tags * 8 bytes + 1k counters * 4 bytes = 12K

Throughput
The maximum intermediate aggregator may take 3 requests during 30 seconds. The size of the request is 12K and network latency could be 120us.

The single node can process all view events. But we need at least 2 machines for redundancy.

Final aggregator.

Memory
The same memory size as the Intermediate aggregator.

Throughput

The same as for the Intermediate aggregator but we need to save the top 1000 topics in the database every 30 seconds. It adds additional 50 ms to the throughput for every 30 seconds.

The single node can process all requests.