**System Design Interview : Top K Heavy Hitters**

These are very interesting problems here

-

Problem here

* Today we design a system for identifying the top k heavy hitters (also known as the top k most frequent items). It’s a popular system design interview question as it may take many different forms. Let’s name a few. For example, we may be asked to find 100 of the most searched keywords on Google or viewed videos on Youtube or played songs on Spotify or shared posts on Facebook. An interviewer may be interested to know the most retweeted tweets on Twitter or liked photos on Instagram.

We don’t want to scan the entire db all the time each time we do top 100 k this is given here.

* Ok, sounds like a typical big data processing problem and MapReduce might help. MapReduce will indeed help us to solve this problem, but MapReduce along is not enough. We need our solution to return list of heavy hitters as close to real time as possible. For example, we need to return a list of most viewed videos for the last several minutes. Which moves this problem to the category of stream processing problems.

Table of Contents

* [Functional Requirements](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Functional_Requirements)
* [Non-Functional Requirements](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Non-Functional_Requirements)
* [Hash table, single host](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Hash_table_single_host)
* [Hash table, multiple hosts](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Hash_table_multiple_hosts)
* [Hash table, multiple hosts, partitioning](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Hash_table_multiple_hosts_partitioning)
* [Count-min sketch](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Count-min_sketch)
* [High-level Architecture](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#High-level_Architecture)
* [Data flow, fast path](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Data_flow_fast_path)
* [Data flow, slow path](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Data_flow_slow_path)
* [MapReduce jobs](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#MapReduce_jobs)
* [Data Retrieval](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#Data_Retrieval)
* [What else](https://serhatgiydiren.com/system-design-interview-top-k-problem-heavy-hitters/#What_else)

### **Functional Requirements**

* topK(k, startTime, endTime) : We want our system to return a list of k most frequent items. And because this list changes over time, we also need to provide a time interval, and more specifically, start and end time of that interval.

### **Non-Functional Requirements**

* Scalable (scales out together with increasing amount of data : videos, tweets, posts, etc) : We want to design a solution that can scale together with the data increase.
* Highly Available (tolerates network / hardware failures, no single point of failure) : Make data available in case of hardware failures or network partitions.

Highly performant system here

* Highly Performant (few tens of miliseconds to return top 100 list) : Fast, so that we can retrieve a list of heavy hitters in a matter of tens of milliseconds. Performance requirement should give you a hint that the final top k list should be pre-calculated and we should avoid heavy calculations while calling the topK API.
* Accuracy (as accurate as we can get) : Without this requirement, we may design a solution that is scalable, highly available and performant, but never produces accurate results. For example, by using data sampling we may not count every event, but only a small fraction of all events. And even though this is a good option to discuss with the interviewer, let’s try to design a solution that can produce accurate results.

**The native solution here**

### **Hash table, single host**

A diagram of a number

Description automatically generated

* Given a list of events, how do we calculate the k most frequent elements?

**Why do we want to use heap instead of sorting the map for storing top k?**

* First, we calculate how many times each element appears in the list. So, we create a hash table that contains frequency counts. And to get top k elements we can either sort the hash table based on frequency counts.  
    
   Or we can add all elements into a heap data structure. We make sure heap contains only k elements every time we add new element to the heap.   
    
  Heap approach is faster. When we sort all elements in the hash table, the time complexity of such algorithm is nLog(n), where n is the number of elements in the hash table. By using heap, we can reduce time complexity to be nlog(k).

Porlbem with this approach:

* The first and the most obvious problem with this solution – it is not scalable. If events are coming with a high rate, single host will quickly become a bottleneck. So, we may want to start processing events in parallel. How do we achieve this?

### **Hash table, multiple hosts**

A diagram of a computer program

Description automatically generated

* One of the options is to introduce a load balancer. This may be a classic load balancer or a distributed queue. Each event then goes to one of the hosts in a cluster. Let’s call them Processor hosts. And because the same video identifier may appear on different Processor hosts, each Processor needs to flush accumulated data to a single Storage host.
* It is a bit better now, we can process events in parallel. Total throughput of the system has increased. But another evident problem of this solution is memory. We may use too much memory on each Processor host as well as Storage host. There are billions of videos on Youtube. Even if we store a fraction of this number in memory, hash table will become huge. What can we do?

### **Hash table, multiple hosts, partitioning**

A diagram of a data parterre

Description automatically generated

* If you recall our previous video where we designed a distributed cache, you will notice that we solved a similar problem there. And the way we did it is by partitioning data into smaller chunks. Let’s apply the same idea here.
* We introduce a new component, called Data Partitioner. And this component is responsible for routing each individual video identifier to its own Processor host. So, each Processor host only stores a subset of all the data. And we follow the same procedure as we did for a single host.

**apply merge n sorted lists**

* We build a hash table, create a heap and add all elements from the hash table to the heap. Now, each Processor host contains its own list of k heavy hitters. And each such list is sorted. How do we create a final list that combines information from every Processor host? And I hope you recognize another classic coding interview question that asks to merge n sorted lists.
* It is important to note that Processor hosts only pass a list of size k to the Storage host. We cannot pass all the data, meaning that we cannot pass each Processor hash table to the Storage host, as one combined hash table may be too big to fit in memory. That was the whole point of data partitioning after all, to not accumulate all the data on a single host.

What if we just want for a 1-hour window or 1 day period?

* But if we want to know heavy hitters for 1-hour or 1-day period, how can we build 1-hour list from 60 1-minute lists? If you think about this for a moment, you will see that there is no way to solve this problem precisely. We need the whole data set for that hour or that day to calculate a list of top k heavy hitters.
* So, on one hand we need the whole data set for a particular time period, let’s say 1-day. And on the other hand, we cannot accumulate data in memory for the whole day. What should we do? Let’s store all the data on disk and use batch processing framework to calculate a top k list.
* This is where MapReduce comes into play. Yes. Another problem with this architecture, is that although it may seem simple, it is not. Every time we introduce data partitioning, we need to deal with data replication, so that copies of each partition are stored on multiple nodes. We need to think about rebalancing, when a new node is added to the cluster or removed from it. We need to deal with hot partitions.

### **Count-min sketch**

A screenshot of a computer

Description automatically generated

Count min sketch

**Why do we use count min sketch?**

- Because using this the size of the hash table will not grow here, so no need to worry about the memory consumption then, this is very important here.

This data structure is called count-min sketch. Count-min sketch is really simple. You can think of it as a two-dimensional array. We define width of this array and its height. Width is usually in thousands, while height is small and represents a number of hash functions, for example 5. When new element comes, we calculate each hash function value and add 1 to the correspondent cell. For example, video A comes, we calculate five hash functions based on video A identifier and put 1 to each of 5 cells. When another A comes we increment each cell value. And repeat the same for one more A. Then B arrives and we add 1 to each of B cells. Hash function H1 produced a collision and we incremented A’s value as well. Then C arrives. And produces collisions with both A and B. Here is how we add data to the count-min sketch.

* How do we apply count-min sketch to our original problem? Roughly, think of a count-min sketch as a replacement of the hash table we had in all our previous discussions. We still need a heap to store a list of heavy hitters. But we replace a hash table, that could grow big, with a count-min sketch that always have a predefined size and never grow in size, even when data set size increases.

### **High-level Architecture**

A diagram of a diagram

Description automatically generated

* Every time user clicks on a video, request goes through API Gateway, component that represents a single-entry point into a video content delivery system. API Gateway routes client requests to backend services. Nowadays majority of public distributed systems use API Gateways, it’s a widely spread practice.

Uisng logging to log the frequency and build the map

* We allocate a buffer in memory on the API Gateway host, read every log entry and build a frequency count hash table we discussed before. This buffer should have a limited size, and when buffer is full, data is flushed. If buffer is not full for a specified period of time, we can flush based on time.

Partion the data

* Initially aggregated data is then sent to a distributed messaging system, like Apache Kafka. Internally Kafka splits messages across several partitions, where each partition can be placed on a separate machine in a cluster. At this point we do not have any preferences how messages are partitioned. Default random partitioning will help to distribute messages uniformly across partitions.  
    
  The data can be portioned using the video id, or random partinoing
* Now the more interesting part starts. We will split our data processing pipeline into two parts: fast path and slow path.
* still need accurate results and data set is big, Hadoop MapReduce should be our choice.

### **Data flow, fast path (this is the approximate path)**

A diagram of a diagram of a cylindrical object

Description automatically generated

* Users click on videos and each view video request is sent to one of the API Gateway hosts. Let’s say user 1 opened videos A, B and C.
* While User 2 opened videos A and C. Video A was opened multiple times by both users. Each request landed on one of the API Gateway hosts and information about this was captured in log files. Information about views is then aggregated on the host and sent to the Distributed Messaging System. And when I say aggregated I mean we just build a hash table on each host.

**The sliding window here:**

* For each video we count how many times it was viewed in the last several seconds. And remember that this hash table is relatively small. Whenever size of the hash table reaches some limit or specified time elapses, we send data out and free the memory for the next hash table. So, the first API Gateway host will send out information about videos A and B. The second host, information about videos A and C. And the third host, information about video A. The first Fast Processor host will pick up and process the first message. And this is how count-min sketch will look like. The second Fast Processor host will pick up messages 2 and 3 and add information to its own count-min sketch. And after aggregating information for several seconds, each Fast Processor host sends count-min sketch information to some Storage host, where we build a single count-min sketch. Merging several count-min sketches together is as simple as summing up values in corresponding cells.

### **Data flow, slow path (the exact path here)**

A diagram of a data flow

Description automatically generated

* We have Data Partitioner, a component that reads each message and sends information about each video to its own partition. Here, each blue cylinder represents a partition of the Distributed Messaging System.

2 jobs

1. Top k map reduce jobs

2. Frequency count mapReduce job

For example, the first message is scattered around partitions 1 and 2. The second message goes to partitions 2 and 3. The third message goes to the second partition. We have Partition Processors aggregate data for each partition. Each processor accumulates data in memory for let’s say 5 minutes and writes this data to files, which are stored in the Distributed File System.  
  
 Frequency Count MapReduce job reads all such 5-minute files and creates a final list per some time interval, let’s say 1 hour. Top K MapReduce job then uses this information to calculate a list of k heavy hitters for that hour.

### **MapReduce jobs**

A diagram of a graph

Description automatically generated with medium confidence

* In MapReduce data processing is split into phases. The input to a MapReduce job is a set of files split into independent chunks which are processed by the map tasks in a parallel manner. First, the record reader parses the data into records and passes the data to the mapper, in the form of a key/value pair. In the mapper, each key/value pair from the record reader is transformed into zero or more new key/value pairs, called the intermediate pairs.

Can you summarize the process?

Partiition the data, find top k list of each partition and merge all into 1 list

The Top K MapReduce job takes this data, splits it into chunks and sends each chunk to a mapper that calculates local top k list. Then all the individual top k lists are sent to the single reducer that calculates the final top k list. As you may see, the Top K MapReduce is based on exactly the same idea we discussed before: partition the data, find top k list for each partition and merge individual lists into the one final list.

### **Data Retrieval**

A diagram of a storage system

Description automatically generated

* We have talked about data ingestion or how data gets into the system. Let’s now talk about data retrieval.
* First of all API Gateway should expose topK operation. API Gateway will route data retrieval calls to the Storage service. And the Storage service just retrieves the data from its underlying database. This requires some more explanation. Let’s say our fast path creates approximate top k list for every 1-minute interval. And our slow path creates precise top k list for every 1-hour interval, 24 lists total for a day. When user asks for the top k list for the last 5 minutes, we just merge together 5 1-minute lists. And get the final list, which is also approximate. When user asks for the top k list for some 1-hour interval, for example from 1 pm till 2 pm, it is easy, we have precise lists calculated for each hour of the day. But when user asks for the top k list for some 2-hour interval, for example from 1 pm till 3 pm, we cannot give back a precise list. The best we can do is to merge together 2 1-hour lists. And results may no longer be 100% correct. This is a bit tricky, I agree. Hopefully, it makes sense.

### **What else**

* API Gateway does a lot of work. Authentication, SSL termination, rate limiting, request routing, response caching are among the few. And API Gateway hosts indeed may not have enough capacity to run any data aggregation processes in the background. But no matter how busy these hosts are, there will be a process that offloads log files from these hosts. Log files are usually sent to some storage or to a separate cluster where they can be further processed. So, we can run our log parsing and aggregation process on that cluster. The rest of the data processing pipeline stays the same.
* There are several alternatives to count-min sketch algorithm. There are counter-based algorithms, like Lossy Counting, Space Saving and Sticky Sampling. There are also modifications of the count-min sketch algorithm. As you may see, the value of k plays its role in several places. On the fast path we merge top k lists together on data retrieval. On the slow path, in Top K MapReduce job, all mappers send local top k lists to a single reducer. And what it tells us is that k cannot be arbitrary large. Several thousands are probably ok, but tens of thousands may start causing performance degradation. For example, during data retrieval when we merge such long lists on the fly. We should also remember about network bandwidth and storage space if we need to support larger values of k.
* The architecture we built with you is not a novel idea. It is also known as Lambda Architecture. Please do not confuse with AWS Lambdas. Lambda Architecture is an approach to building stream processing applications on top of MapReduce and stream processing engines. We send events to a batch system and a stream processing system in parallel. And we stitch together the results from both systems at query time. In 2011, Nathan Marz, the creator of Apache Storm wrote a famous article called “How to beat the CAP theorem” where he introduced the idea of Lambda Architecture. There are many benefits of Lambda Architecture, but there are drawbacks as well. In 2014, Jay Kreps, one of the authors of Apache Kafka wrote an article where he questioned Lambda Architecture. And he mentioned complexity as one of the main drawbacks of this approach. And indeed, if you look at the fast path once again, you will see that it is very simple. But by introducing the slow path, we greatly increased overall complexity of the system. Yes, we did this to have accurate results and to aggregate data over long time intervals. But these benefits come with a price. And this price is complexity.
* As always, everything is a trade-off in the world of system design. And I bet you are wondering right now, why on earth I want to make your life more complicated. Couldn’t we just use Kafka plus some stream processing framework, for example Apache Spark and solve this problem having a simpler architecture. Yes and no. Depending on the data volume we can solve the top k problem precisely and having a single data processing path. And Kafka + Spark can do the trick. But I wanted to show you stuff under the hood. At the end of the day Spark internally relies on the same ideas we discussed: data partitioning and in-memory aggregation. Spark internally partitions the data and calculates a top k list for each partition using a heap data structure. And all these lists are merged together in the reduce operation.
* Today we solved a tough problem. Problem that requires knowledge (remember count-min sketch data structure, MapReduce paradigm, principles of data aggregation). But we did even more than this, we designed a solution for a whole class of problems. So, in your interview, when you are asked explicitly about identifying heavy hitters, you now know where to start, right? But an interview question may be stated differently and solution we built today still applies. For example, you may be asked to design “what’s trending” service. Google trends, Twitter trends, Youtube trends, etc. In reality these services are more complicated than just a list of heavy hitters, but a simpler version of such service may purely rely on our today’s design. We may compute a list of popular products based on product page views or product purchases. If asked about identifying the most actively traded stocks, this should also ring a bell in your head. Basically, we have a stream of stock trades as an input. And we need to find the top k stocks with the most number of trades. You may be asked to design a system that protects from DDoS attack. And if you think about this problem you will see that this is also a problem of identifying heavy hitters, where heavy hitter is some IP address that floods the victim service with superfluous requests. We just need to find the top k IP addresses that generates the most number of requests. And block those IPs. So, knowing how to solve the top k problem will help you better deal with several other system design questions.