I. Introductory

Hi. Welcome to the system design interview channel. 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.

And I am sure you can come up with more examples. You probably have a feeling already that database or distributed cache, when applied directly, is not a good option for services of such scale. We are talking about hundreds of thousands requests per second, even millions at peak. Databases today can handle millions of requests per second. But even if we count all the views for Youtube videos in the database for some period of time, how do we calculate the top 100? We need to scan over entries in the database and order entries by view counts.

For small scale such approach is fine, but for services of Youtube scale this is very ineffective: both expensive and slow. 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.

Stream processing problem
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. If you are not familiar with MapReduce paradigm, do not worry, we will discuss it in more details later in this video.

Interviewer is helper
As you may see I demonized the interviewer a bit. Portrayed her like someone who wants to make our life miserable by throwing all those requirements on us. This is far from the truth in real life. And let me explain why.
Interviewer is there to help you succeed and uncover your potential. Period. End of message. This is especially true for interviewers who ask system design questions. Usually, those are seasoned engineers, who demonstrated many leadership qualities throughout their career and became senior engineers for a reason. They know that system design interviews are hard, they may spend months solving the problem they ask you during the interview. They do not expect you to solve this problem in full in 60 minutes. But you should not expect an easy conversation either. Otherwise, what's the point.

Next time you go to an interview, remember me saying that interviewer is there to help you demonstrate your best. Treat your next system design interview as a conversation with your colleague, when both of you have interesting problem to discuss. This is not an exam and there is no one single correct answer in most of the system design interview questions. On this positive note let's move on and define requirements for our system.

II. Requirements

Functional requirements are simple. 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**.

Remember, in our previous video we discussed 3 non-functional requirements we should think about while designing distributed systems. Let's apply them here as well.
So, we want to:
1) design a solution that can **scale** together with the data increase,
2) **make data available** in case of hardware failures or network partitions and
3) **fast**, so that we can retrieve a list of heavy hitters in a matter of tens of milliseconds.
4) **accuracy**

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. Let's also add one more requirement, which is accuracy. 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. As usual, let's play around with some simple ideas and try to get hints from there.

III. Solution

We start with a single host. And let's assume the whole data set can be loaded into a memory of that single host. For example, we have a list of events (video views). Every time user opens a video, we log such event by adding video identifier to the list of events:
A, B, C, D - represent unique video identifiers.

Given a list of events, how do we calculate the k most frequent elements? 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:
1) **sort the hash table** based on frequency counts.
2) 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.
1) When we sort all elements in the hash table, the time complexity of such algorithm is n*log(n), where n is the number of elements in the hash table.
2) By using heap, we can reduce time complexity to be n*log(k).
So:
sort - n*log(n)
heap - n*log(k)

Code
Problem we have just discussed is a classic coding interview problem. It is marked as a medium difficulty problem on the leetcode. Let's take a look at the code.

1. We first define a class that stores video identifier and number of times this particular video was viewed.
list_top_k = TopK(list_of_events)
TopK method:
1) gets a list of events as the input
2) returns a list of k heavy hitters

2. We then create a hash table that counts how many times each video appeared in the list of events.
3. We define a min heap data structure (priority queue in Java). And add each element from the hash table to the heap. While doing this, we check if heap has more than k elements. And if this is the case, we remove a top element from the heap.

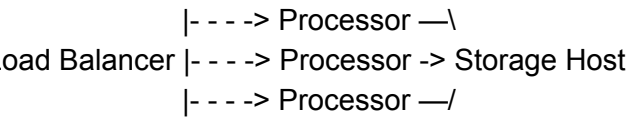
Because this is a min heap, the top element of the heap is the one with the minimum frequency count. Elements with higher frequency count remain in the heap, while elements with lower frequency count are removed periodically. This way we make sure that only heavy hitters remain in the heap.

4. And the last step is to prepare a result list. Heavy hitters will be ordered from minimum to maximum frequency count in this list.

IV. Scalability problem

1) Speed

Single host solution was easy to build, right? But 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 <----------Because не успевает/мало памяти - про это?
=>So, we may want to start processing events in parallel.



How do we achieve this? 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.
(не совсем понял. Мы же распределяем события по Processor. Одно событие попадёт только к какому-то Processor. А, имеется в виду, что тип события “C”(к примеру) может и в тот процессор попасть и в другой.)

2) Memory

(hash table will become huge -> partitioning data into smaller chunks)
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.
(т.е. вначале говорилось о скорости обработки, всё-таки)
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? 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.

Data Partitioner
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.
(как и было раньше так-то. А, нет. Он типы событий группирует в subsets(1: A,B,C, 2: E,D,F) и отправляет события Процессору, который отвечает(забиндин) за данный сабсет(подмножество).
So, each Processor host only stores a subset(о, вот!) of all the data. And we follow the same procedure as we did for a single host. 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.
(вот это что-то новое. Отличие в предварительном подсчёте topK каждым Processor)

Merge n sorted lists
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. If you are unfamiliar with this problem, please check a link in the description to this video.

Point to not accumulate all the data on a single host

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. Ok, we took one more step in improving our solution.

=>By partitioning the data, we increased both scalability and throughput.

Streaming data is unbounded

This architecture is not bad, but it has some serious drawbacks and needs further improvement. Let me explain. All this time we were talking about bounded data sets or data sets of limited size.

Such data sets can indeed be split into chunks, and after finding top k heavy hitters for each chunk we just merge results together. But streaming data is unbounded, essentially infinite.

Users keep clicking on videos every second. In these circumstances, Processor hosts can accumulate data only for some period of time, let's say 1-minute, and will flush 1-minute data to the Storage host.

Processor Storage

-----flush topK data-----> (0;1m) data

1m

-----flush topK data-----> (1;2m) data

...

Cannot afford storing information about every video in memory. But for 1 hour topK need it

So, the Storage host stores a list of heavy hitters for every minute. And remember that this is only top k heavy hitters. Information about all other elements, that did not make to the top k list is lost. We cannot afford storing information about every video in memory. Great, 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?

(Just to make hash table. Put data form all 1-minutes lists. For example, put data of 0-1m list(B=2,A=3) to the hash table. Put 1-2m list(C=2,B=2). So in the hash table A=3, B=4, C=2. The only drawback is that this hash table can grow over memory.)

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. I will greatly appreciate if someone can provide a simple explanation of this fact in the

comments section.

Store all the data on disk + MapReduce
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. Aha, this is where MapReduce comes into play. Yes.

Problems of replication, copies of each partition, rebalancing are not critical
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. All these problems are not critical and solutions are known.

V. Simple solution

Simple solution - sacrifice is accuracy - data structure with fixed size memory
But before diving into all these complexities let's ask ourselves: is there a simpler solution to the top k problem. And the answer is yes, there is a simpler solution. But we will need to make some sacrifices along the way. And the main sacrifice is accuracy.

Let's take a look at some amazing data structure that will help identify a list of heavy hitters using fixed size memory. But results may not be 100% accurate. Bear with me, we are very close to outlining the final architecture. This data structure is called count-min sketch.

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.

But how do we retrieve data? Logic is simple, among all the cells for A we take the minimum value. Because of collisions, some cells will contain overestimated values. And by taking minimum we decrease a chance of overestimation. And, hopefully, it makes total sense right now why we need several hash functions, and not a single function. If it was just a single hash

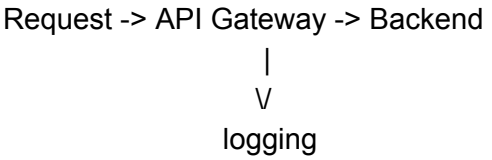
function (for example H1), meaning that our array had only a single row, value of A would be 5, instead of 3.

By using more functions, we decrease the error. And I bet you wonder how we choose width and height parameters.

Count-min sketch is a very well-studied data structure. There are formulas that help to calculate both parameters based on the accuracy we want to achieve and probability with which we reach the accuracy. Great, but how do we apply count-min sketch to our original problem?
Glad you asked. 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.

Replace a hash table -> count-min sketch with predefined size
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. We have armed ourselves with many ideas by now. Let's put everything together and outline the high-level architecture.

VI. High-level architecture



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.

Log generation
For our use case, we are interested in one specific function of API Gateways, log generation, when every call to API is logged. Usually these logs are used for monitoring, audit, billing.
We will use these **logs for counting** how many times **each video was viewed**.

Background process
We may implement a background process that:
1) reads data from logs,
2) does some initial aggregation, and
3) sends this data for further processing.

1) We allocate a buffer in memory on the API Gateway host,
2) read every log entry and

3) 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.

There are also other options, like aggregating data on the fly, without even writing to logs. Or completely skip all the aggregation on the API Gateway side and send information about every individual video view further down for processing.

Data into a compact binary format
There are pros and cons for every option. Will return to this topic a bit later. Last but not least, we better serialize data into a compact binary format. This way we save on network IO utilization if request rate is very high. And let CPU pay the price. Once again, all these considerations depend on what resources are available on the API

Gateway host:
1) memory
2) CPU
3) network or disk IO.
Great stuff to discuss with the interviewer.

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. Pretty standard setup, right?

Fast path and slow path
Now the more interesting part starts. We will split our data processing pipeline into two parts: fast path and slow path.

1) On the fast path, we will calculate a list of k heavy hitters approximately. And results will be available within seconds.
2) On the slow path, we will calculate a list of k heavy hitters precisely. And results will be available within minutes or hours, depending on the data volume.

1) Fast path

Let's first look at the fast path, it is much shorter. We have a service, let's call it fast processor, that creates count-min sketch for some short time interval and aggregates data. And because memory is no longer a problem, we do not need to partition the data. Any message from Kafka can be processed by any Fast Processor host.

Every time we keep data in memory, even for a short period of time, we need to think about data replication. Otherwise, we cannot claim high availability for a service, as data may be lost due to

hardware failures. Because count-min sketch already produces approximate results, meaning we lose data already in some way, it may be ok if we lose data in some rare cases when host dies.

As long as slow path does not lose data, we will have accurate results several hours later. Absence of replication greatly simplifies the system. A good trade-off to discuss with your interviewer.

Every several seconds Fast Processor flushes data to the Storage. Remember that count-min sketch has a predefined size, it does not grow over time. Nothing stops us from aggregating for longer than several seconds, we may wait for minutes. But this will delay final results. Another trade-off to discuss.

Storage
The Storage component is a service in front of a database. And it stores the final top k list for some time interval, for example every 1 minute or 5 minutes. Have you noticed how our data processing pipeline gradually decreases request rate?

Let me elaborate on this. There may be millions of users clicking on videos in the current minute. All those requests end up on API Gateway hosts. This cluster may be big, it may contain thousands of hosts. Then we pre-aggregate data on each host, may be just for several seconds. But it helps to decrease number of messages that go into Kafka.

Next is Fast Aggregator cluster. And it will be much smaller in size than API Gateway cluster. It aggregates data further. For another several seconds. And when we arrive to the Storage, we only deal with a small fraction of requests that landed initially on API Gateway hosts. This is an important concept when you deal with stream processing and data aggregation.

Please have it in mind for other system designs. Now, let's define components of the slow path.

2) Slow path

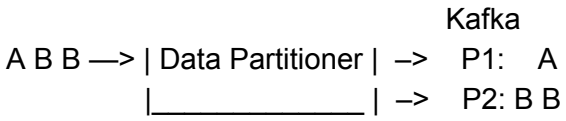
On the slow path we also need to aggregate data, but we want to count everything precisely. There are several options how to do this. One option is to let MapReduce do the trick.

MapReduce
1) distributed file system
2) job to calculate frequency counts
3) calculate the actual top k list
We dump all the data to the distributed file system, for example HDFS or object storage, for example S3. And run two MapReduce jobs, one job to calculate frequency counts and another job to calculate the actual top k list. Data is then stored in the Storage service. I feel like something is missing here.

Count-min sketch is great, it helped us to create a very simple and cost-effective architecture. But results produced by our fast path are approximate. MapReduce jobs we introduced on the slow path will help to produce accurate results. But this data processing path is indeed slow. I wish we could have something faster than MapReduce jobs.

It may be slower than count-min sketch solution, but still accurate. Let's return to the idea of data partitioning we discussed in the past.

Data Partitioner

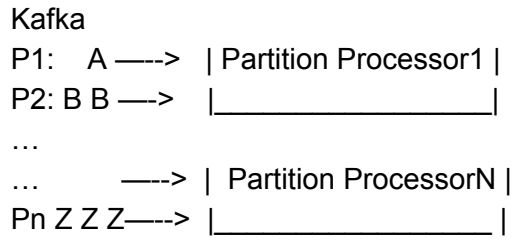


Let's Data Partitioner read batches of events and parse them into individual events. Partitioner will take each video identifier and send information to a correspondent partition in another Kafka cluster.

Hot partitions, data replication

And as we mentioned before, every time we partition data, we need to think about possibility of hot partitions. And our partitioner service should take care of it. Now, each partition in Kafka or shard in Kinesis, depending on what we use, stores a subset of data. Both Kafka and Kinesis will take care of data replication.

Partition Processor



And we need a component that will read data from each partition and aggregate it further. Let's call it a Partition Processor. It will:

1) aggregate data in memory over the course of several minutes
2) batch this information into files of the predefined size
3) send it to the distributed file system, where it will be further processed by MapReduce jobs.

3) Medium path

Partition Processor may also send aggregated information to the Storage service. Wait, wait, wait. Send data to the Storage service from three different places? I may not clearly explain it yet, but I feel that there is an overlap somewhere. Is this what you are thinking right now? Your concern is valid. Ideally, we should have a single data processing path. I just wanted to show

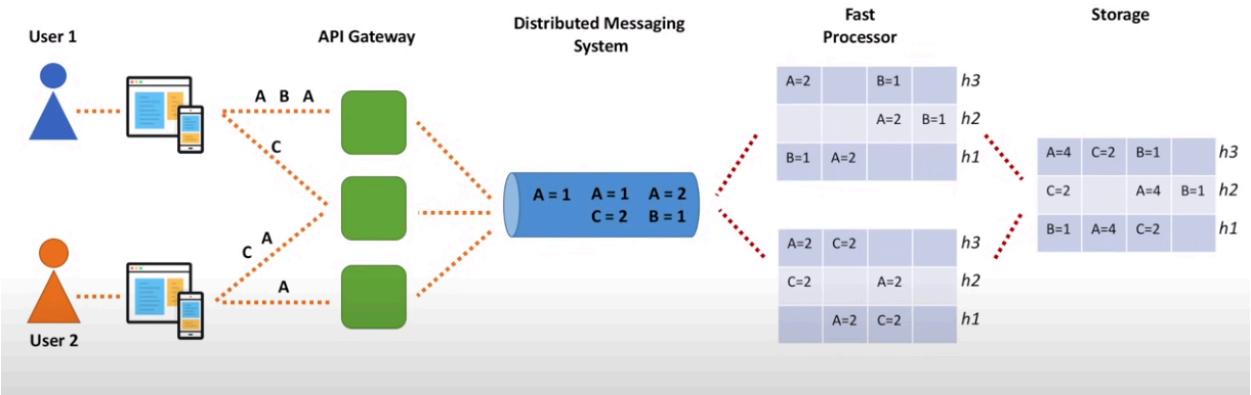
you that depending on the requirements and more specifically how soon top k heavy hitters should be identified, we may choose a path that suits us the best.

What path to choose?
1) Need fast calculation, no accuracy -> Fast path
For example, if we can tolerate approximate results, forget about the slow path and its components. Fast path is simple and should be cheap to build.
2) Need medium(minutes) calculation with accuracy -> Medium path
If accuracy is important and results should be calculated in a matter of minutes, we need to partition the data and aggregate in memory.
3) Need calculation of big time ranges with accuracy -> Slow path
And if time is not an issue but we still need accurate results and data set is big, Hadoop MapReduce should be our choice. Let me walk you through a simple simulation, that will help you better understand the meaning of each component in the architecture.

VII. Work flow

1) Fast path

We will start with the fast path. 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.



Aggregating views on API Gateway instances

```
|-------API Gateway-------|
|                         |
|     |predefined small MB|   |
|     |            hash table|   |
---------------------------------
```

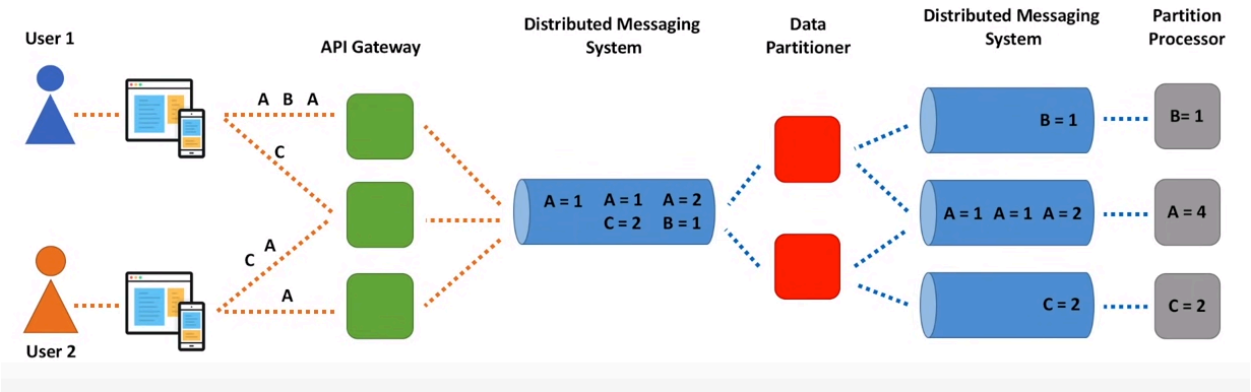
And when I say aggregated I mean we just build a hash table on each host. For each video we

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
1) The first API Gateway host will send out information about videos A and B.
2) The second host, information about videos A and C.
3) 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.

1) Slow path

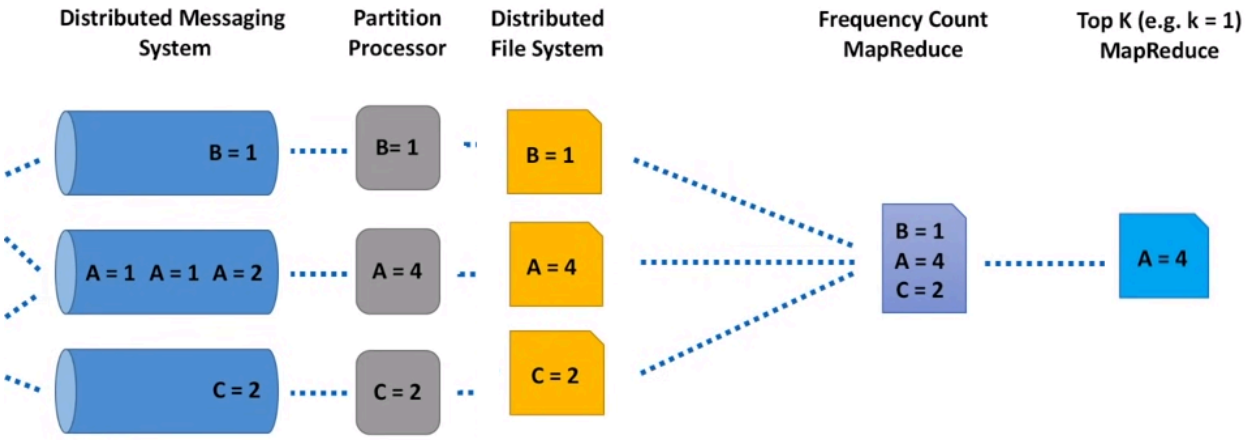
Now, let's take a look at the slow path.



We have:
Data Partitioner - a component that:
-----> 1) reads each message
2) sends information about each video to its own partition ----->

Distributed Messaging System
Here, each blue cylinder represents a partition of the Distributed Messaging System. 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.

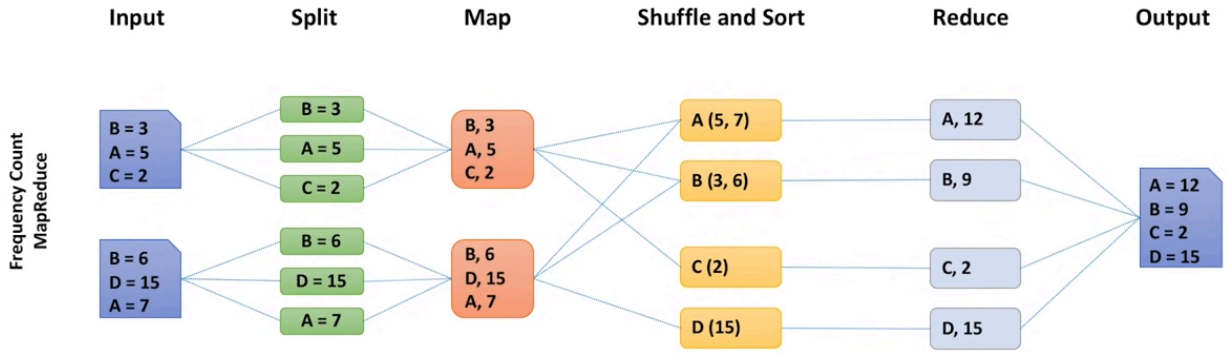
Partition Processors
We have Partition Processors aggregate data for each partition. Each processor accumulates data in memory for let's say 5 minutes and writes this data to files, which are stored in the Distributed File System:



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.

Frequency Count MapReduce

And let's take a bit closer look at MapReduce jobs. 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.

1) First, the **record reader:**

a) parses the data into records and

b) passes the data to the mapper, in the form of a key/value pair.

2) In the **mapper**,

a) each key/value pair from the record reader *is transformed into* zero or more new key/value pairs, called the intermediate pairs.

3) In the next phase, the **partitioner:**

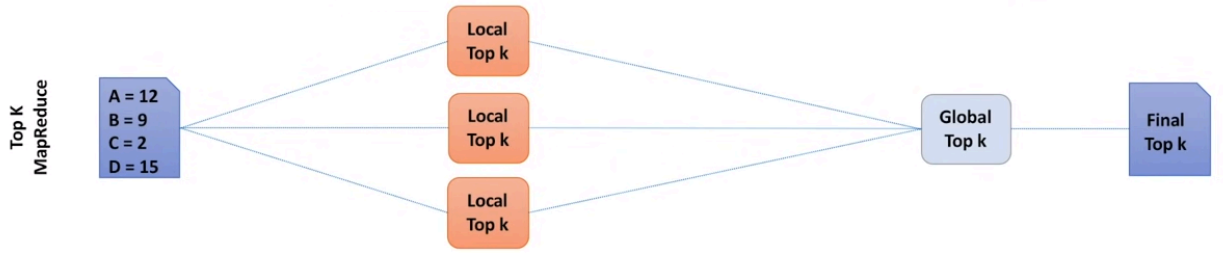
a) takes the intermediate key/value pairs from the mapper and

b) assigns the partition.

Partitioning specifies that all the values for each key are grouped together and make sure that all the values of a single key go to the same reducer. The partitioned data is written to the local

disk for each mapper.
4) During the shuffling phase,
Hadoop MapReduce takes the output files produced by partitioners and downloads them to the reducer machine, via HTTP. And the purpose of the sort is to group keys together.
This way reducer task can easily iterate over values. Reducer then takes each key and iterates over all of the values associated with that key. In our case we just sum up all the values for the key. And the output of the reduce task is written to the file system.

Top K MapReduce job



The Top K MapReduce job:
1) takes this data
2) splits it into chunks
3) 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:
1) partition the data
2) find top k list for each partition
3) merge individual lists into the one final list.
So far, we have talked about data ingestion or how data gets into the system.

Data retrieval
Let's now talk about data retrieval. If you remember, it was our functional requirement to retrieve a list of k heavy hitters for a specified time interval. Here is what we should do.

First of all API Gateway should expose topK operation.
1) API Gateway will route data retrieval calls to the Storage service.
2) 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.

User behavior/scenarios/cases

1) 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.
2) 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.
3) 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.

VIII. Interviewer questions

Let's see what else an interviewer may ask us about.

API Gateway does a lot of work:
1) Authentication
2) SSL termination
3) rate limiting
4) request routing
5) 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:
1) to some storage
or
2) 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.

Alternatives to count-min sketch
There are several alternatives to count-min sketch algorithm:
1) counter-based algorithms, like Lossy Counting
2) Space Saving
3) Sticky Sampling
There are also modifications of the count-min sketch algorithm. If you are further interested in sketching algorithms, please check online or send me a message. This area is well-studied and I will share some good materials with you. As you may see, the value of k plays its role in several places.

Single reducer
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. Just keep this in mind. This is a really good question.

Lambda Architecture = building stream processing applications on top of MapReduce
What do you think? Please pause this video and try to think what are the drawbacks of the system we just designed. 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.

Slow path increase overall complexity of the system
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.

We are almost done for today. To those of you that have made it this far, you deserve praise. And let me share with you some important observation. 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. And that is it for today. Thank you for being with me. Next time let's talk about distributed counters.
When we need to count page views at Facebook scale or ad views (impressions) at Google scale. See you soon.