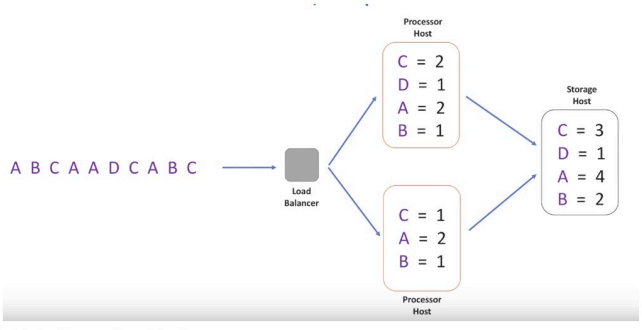
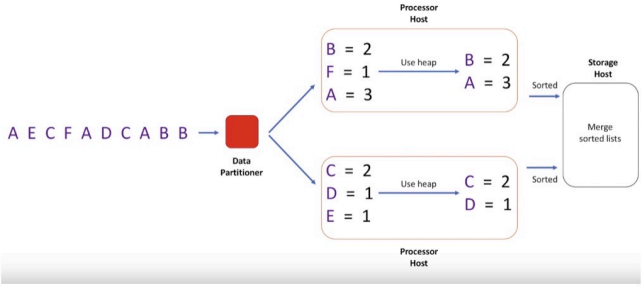
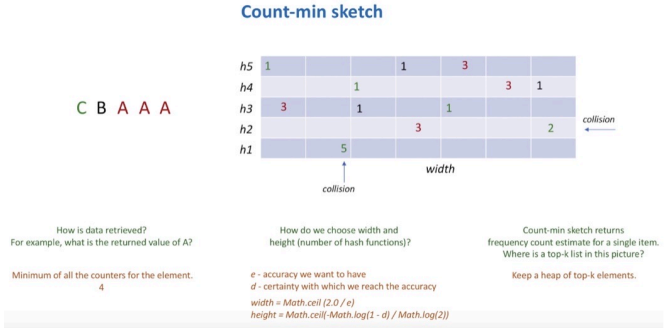
- **Use Cases**
  - Find 100 most
    - Viewed videos on youtube
    - Played songs on Spotify
    - Shared posts on facebook.
    - Retweeted tweets on twitter.
    - Liked Photos on Instagram
    - Searched keywords on Google.
- With Such a scale, database or distributed cache is not an option. We might be dealing with 1M RPS. If we would use DB to track view counts, first the writes/updates would be super slow, and then finding the top K items would require scanning the whole dataset.
  - May be MapReduce can help. But it is not sufficient. We need to return heavy hitters in as close to realtime as possible.
    - e.g. Calculate top 100 list for last
      - 1 min, 5 min, 15 mins, 60 mins etc.
    - This makes this problem in the flavors of **stream processing** problem.
- **Requirements**
  - **Functional**
    - topK(k, startTime, endTime)
  - **Non-Functional**
    - Scalable(Scales together with increasing amount of data.)
    - Highly available(survives hardware/network failures, no SPOF)
    - Highly Performant(few 10s of ms to return top 100 list)
      - Given the performance requirement, it is a hint that the final list should be pre-calculated and we should avoid heavy calculations while calling the top K API.
    - Accurate
      - For e.g. by using data sampling, we may not count every element, but only a small fraction of events.
- **Approaches**:
  - Hash table, Single Host.
    - Keep the count of the incoming list of events in a hashmap.
      - 2 Approaches.
        - Sort the list of entries in the hashmap by frequency and return the first K elements. Time O(nLogn)
        - Put the elements on a heap of size K. Time-O(nLogK)
    - This is not scalable as the volume of events incoming goes too high. We may need to process events in parallel.
  - HashTable, multiple Hosts.
    - Memory would be a problem if you would store all the youtube videos IDs in the memory you the host.



  - HashTable, Multiple Hosts, Partitioning.
    - We won't send in all the hash table data from all hosts to the storage host. Instead we would compute topK list individually at each host, and we need to merge these sorted list finally on the storage host.

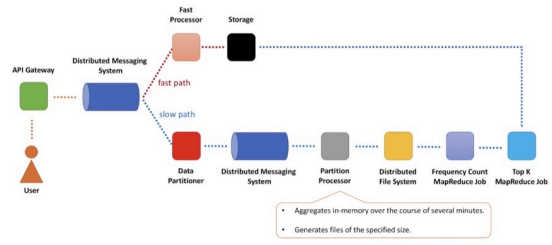


- **Problem?**
  - We considered the data set to be unbounded, thats why we were able to think about partitioning it into multiple chunks. But streaming data isn't bounded. It keeps on coming. In this case, the processor host can keep on accumulating the data only for certain period of time, before which it will run out of memory. Say 1 min.
    - We will flush 1 min data to the storage host.
    - Storage host stores the list of heavy hitters for every minute.
    - We are intentionally losing all the information about non-topK elements. We can't afford storing information about every video in memory.
    - But what if we want to find topK in last 1 hour, or last 1 day, how can we build it using 60 1 min list??
      - Given the current approach, there isn't a correct way to solve this problem. To find the topK for the day, we need full dataset for the whole day.
    - Conflicting requirements, keep whole 1 day data(to satisfy the requirement) or lose it to afford storage.
      - Let us store all the data on the disk since it can't fit into the memory, and use batch processing frameworks to do topK lists.
        - Map Reduce architecture would come into play.
  - Another problem with this arch is even though it may seem simple, it is not. Think about?
    - Every time we introduce data partitioning, we have to think about data replication so that copies of each partition are stored on multiple nodes.
    - We need to think about re-balancing, when a new node is added/removed to/from the cluster.
    - We need to deal with hot partitions.
  - Before jumping into the above discussed approaches, let us think if there is a simple solution to the topK problem?
    - Yes, but we need to make sacrifices along the way. Accuracy is the sacrifice.
    - Data Structure that would help us count topK with fixed size memory, but results may not be 100% accurate.
      - **Count-Min Sketch.**
        - Think of it has 2 dimensional array.
          - Width is usually in thousands, and height is small(say 5 which is the list of the hash functions.)
      - Whenever a new element comes, we calculate the hash value, and add 1 to the corresponding cell.
      - There could be collision at some places. Thus we take the smallest value across the hash function values for an element.
      - Count-Min sketch is a fixed size data structure, even when dataset size increases.

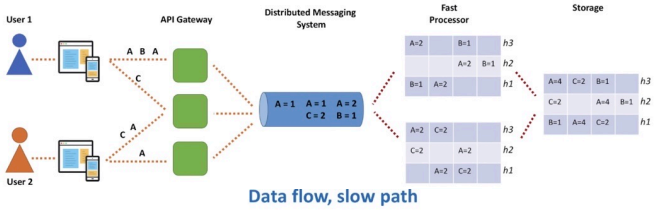


- **High Level Design**
  - **API Gateway** into let us say the video content delivery system which would be serving the video request.
    - For our use case we are interested in 1 function of the API gateway, Log Generation, where every call to the API is logged. Usually these logs are used for monitoring, logging, audit. We will use these logs to count, how many times each video was viewed. We may have a background process that reads data from the logs, does some initial aggregation and sends the data for further processing.
    - Allocate memory for a buffer on the API gateway service, read the log line, and build a frequency count hash table. This buffer should have a limited size, and when the buffer is full, the data is flushed. If the buffer is not full at a period of time, we can flush based on the period of time.
    - Other option could be aggregating data on the fly, w/o being writing to the log file. or completely skip data aggregation on the API gateway side, and send the information about every individual event (video being viewed) further down for processing. Evaluate pros and cons for each option.
    - We can save on Network IO utilization by serializing the data in a compact binary format(e.g. Apache Avro) and Let CPU pay the price. All these considerations would be dependent on what resources are available on API gateway host namely memory, cpu, network, and disk IO.
  - Initial aggregated data is sent to a **distributed messaging system**, like Apache Kafka.
  - Next the we can divide the system into 2 paths, Fast path and Slow path.
    - In Fast path, we will calculate the result of topK hitters approximately. Results will be available within seconds.
    - In slow path, we will calculate the result of topK hitters precisely. Results available within minutes/hours.
  - Depending upon the system constraints on timing of the system(whether near-realtime results are needed or if the delay is acceptable to achieve accuracy), you should choose either path.

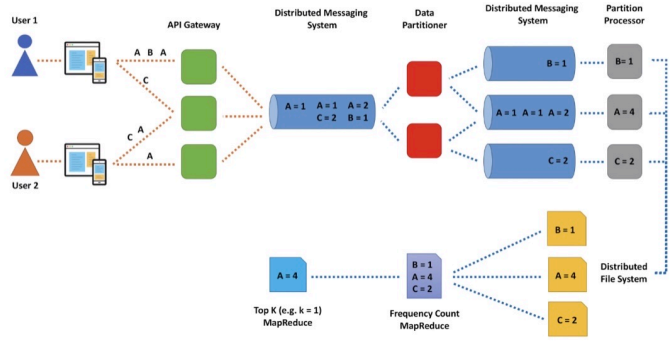
High-level architecture



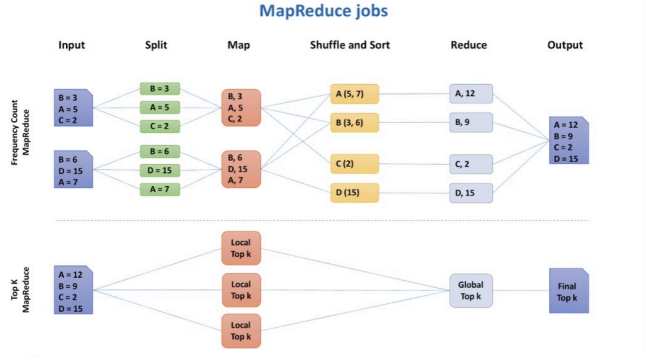
Data flow, fast path



Data flow, slow path

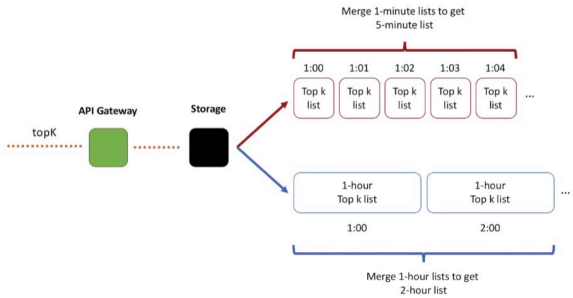


MapReduce jobs



- **Data Retrieval by the client.**
  - Merging 2 different result sets to answer the API call won't be accurate, but that is a tradeoff. You can't aggregate data for any time. You would have to take a buy in on what exactly is needed and build according to that.

Data retrieval



Followup Questions by the I/w?

- What is API gateway doesn't have enough resources to do the data aggregation processes on the host, because it would already be dealing with SSL termination, routing requests, authentication, rate limiting, response caching etc.
    - Log files would still be uploaded from this host to some storage on a different cluster. We can do log parsing on the other cluster. Rest of the processing pipelines are still the same.
- Any alternatives to count-min sketch algorithms?
    - Counter based algorithm like lossy counting, space saving, and sticky sampling.
- How big is K?
    - Value of K as several thousand would still be okay, but if it goes to 10s of thousands, it may causing performance degradation.
        - e.g. on data retrieval, when we merge such lists on the fly.
    - Also, network bandwidth and storage space for larger values of K.
- What are the drawbacks of the architecture?
    - The architecture we build is not a noble idea, and is called Lambda architecture(not to confuse with AWS lambdas). Lambda architecture is the approach of building stream building applications on the top of Map reduce 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.
    - Nathan Marz,2011, Creator of apache storm wrote an article called, "How to beat the CAP theorem" where he introduced the idea of Lambda architecture.

Other application of topK hitters systems?

- Design a "***Whats Trending"*** service.
    - Google, Youtube, Twitter trends etc. Simpler version of such systems can be built using above design.
- Design a system that displays **"List of Popular products."**
- Find most actively traded stocks.
- Design a system that protects from DDos attacks.
    - This is also a problem of identifying heavy hitters.