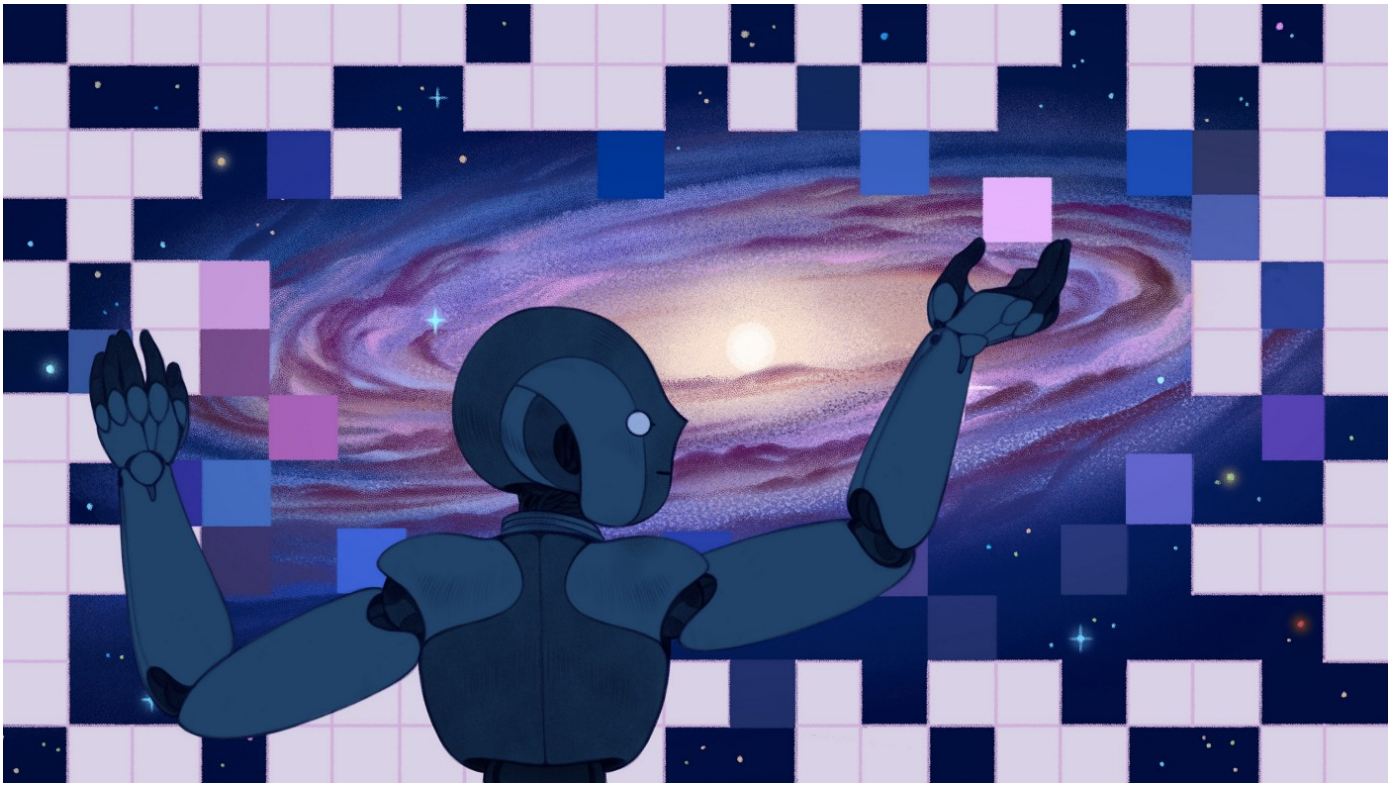
System Design — Top K Trending Hashtags



source: quanta magazine

> *Design a system to compute the top K trending hashtags in the last 24 hours for Twitter/Instagram.*

Similar questions

- Show the top 10 listened songs in the last 7 days per user on a music streaming app.
- Show the top 10 most viewed/clicked news article on LinkedIn in the last 24 hours.
- Show the top 100 most searched results in the last 24 hours on Google.

Functional Requirements

- Users should be able to see the top K trending hashtags in the last 24 hours. Here K can be dynamically updated and/or the time interval can be updated from 24

Non Functional Requirements

- System should be highly available and scalable
- Need not be strongly consistent i.e. eventual consistency
- Results could be approximate but with a very less error rate.
- Results should be returned with a very low latency

Traffic and Throughput Requirements

- 500 million daily active users.
- Assuming 80% of the users write a tweet or re-tweet, number of new tweets per day = 400 million.
- Assuming each tweet has approximately 5 hashtags, number of hashtags per day = 2 billion.
- Assuming that 25% of the hashtags are unique, number of unique hashtags per day = 500 million
- Write Throughput = 2 billion/86400 = 23K hashtags per second
- Assuming that each user refreshes the feed or opens the feed 5 times in a day, number of read requests = 500 million*5 = 2.5 billion
- Read Throughput = 2.5 billion/86400 = 29K reads per second
- p99 latency requirements = 200 ms

The “On My Computer” Approach

(In memory) **Double Ended Queue+Sorted Set**

Maintain a double ended queue for each unique hashtag.

> *Insert the timestamp of the incoming hashtag at the end of the queue. If the time difference between the end and front timestamp of a queue is greater than 24 hours, then pop the front of the queue till the difference is less than equal to 24 hours.*

> *Maintain a sorted set data structure such as AVL Tree (Balanced BST) or Skip List for storing the lengths of the above queues for each hashtag. Whenever a queue is updated, the length is updated in the sorted set in O(logN) time complexity.*

For queries, fetch the highest K values of queue lengths from the sorted set along with the hashtags. Can be done in O(K*logN) time complexity.

Assuming that we are storing the UNIX Timestamps in seconds, thus number of bits required is 32 bit timestamp. Total size of in-memory queue = 2 billion hashtags * 32 bits = 8 GB. Lets assume another 8GB for the in-memory sorted set. Thus total memory required to compute the top K hashtags = 16GB.

16GB is totally manageable size of RAM for most modern hardwares.

But there are few issues:

- Not durable. If the system crashes or restarts all data will be lost.
- Instead of 24 hours, if the requirement is for 30 days=720 hours, then RAM size requirements will be 30 times, which is then not possible on local hardwares.
- Handling high QPS. Number of CPU cores required (approximately) to serve 29K read requests = 29K*200/1000=5800. This many CPU cores is not possible with a single system.

The above problems makes the system non-available and non-scalable (1st non-functional requirement)

Distributed Database/File Storage

In order to persist the data, store the hashtags in a database or a file storage.

We can use a NoSQL database such as Cassandra to store the hashtags along with the timestamp with a schema something like (event_id, hashtag, timestamp) with the event_id as the partition key and the timestamp as the range key so that the data is sorted by timestamp.

Instead of storing the timestamp separately, we can use SnowflakeID format for the event_id which is ordered by the

timestamp.

Assuming that we are storing hashtags for 1 year, the total number of hashtags = 365*2 billion = 730 billion.

Assuming that each hashtag is approximately 7 characters longs i.e. 7 bytes=56 bits, snowflake event_id is 64 bits, thus total storage requirement is 730 billion*120 bits = 10TB.

This much data can be accommodated on a single Cassandra instance.

> *In order to deal with failure of a single node, create a Cassandra cluster with at-least 3 nodes such that same data is replicated across all the 3 nodes for redundancy. This ensures high availability.*

We can choose to go ahead with **asynchronous replication for our cluster since our requirement was eventual consistency**.

> *Instead of an expensive Cassandra instance, we can also choose to use a distributed file storage system such as S3 or HDFS.*

In S3 we can create folders for data for each day for the last 1 year and inside each folder have multiple sub-folders corresponding to each hashtag and inside each hashtag folder we have multiple files corresponding to the hashtag ingested for each hour.

Running Jobs to accumulate hashtag frequencies

> *Run Spark jobs every hour to scan the S3 folders, read the files and sum the number of events and store these results in another database such as MySQL and also in an in-memory cache such as Redis.*

Maintain a MySQL table for Spark jobs to track which S3 file to read next. At the start of each hour, the job reads which S3 file to read for a hashtag and then reads that file and computes the frequency and updates the MySQL table.

(hashtag, frequency, datetime)

Number of rows of MySQL table in 24 hours (assuming 500 million unique hashtags and each hashtag appears every hour) = 500 million*24 = 12 billion.

Assuming each row is of approximately 128 bits, total size of MySQL table = 12 billion*128bits = 180GB.

For fetching the frequencies of the hashtags in the last 24 hours, we make an SQL query:

SELECT hashtag, SUM(frequency) FROM 'ABC' GROUP BY hashtag.

Since the data size is huge, we would like to partition the data by hashtag. Assuming that we have 100 MySQL instances, we maintain a consistent hashing table to map a hashtag to an instance using HASH(hashtag)%100.

Approximately each instance will store 2GB of data. Running the SQL query on 2GB of data will approximately take 20 seconds assuming disk read speed is 100MB/s.

We can store these results in Redis cache because the frequencies will not get updated for a period of 1 hour. To store 180GB in Redis cache we will likely need 12 Redis instances assuming each instance to be 16GB.

To partition the 180GB data into 12 instances we can use HASH(hashtag)%12

But there is one problem here —

> *There could be certain hashtags which is trending in the last 20–30 minutes and those frequencies will not be updated before an hour.*

Redis to the rescue.

Maintain a hash-map in Redis where the key is a hashtag and value is the current frequency.

Whenever a hashtag is ingested through API, apart from being written to S3, it is also written to the Redis hash-map and the frequency is updated in the hash-map.

> *Now for computing the frequencies of the hashtags in the last 24 hours, apart from making an SQL query we also query the Redis hash-map. Take the sum of the frequencies for each hash-map from SQL query and Redis. In this way we can have the most up-to-date frequencies for each hash-tag.*

Now flush the Redis hash-map after each hour else counts will be duplicated.

Size of data in 1 hour = 180GB/24 = 8GB. Thus we can store this hash-map on a single Redis instance.

What happens if any MySQL or Redis instance goes down ? System will not be available anymore.

Have replicas.

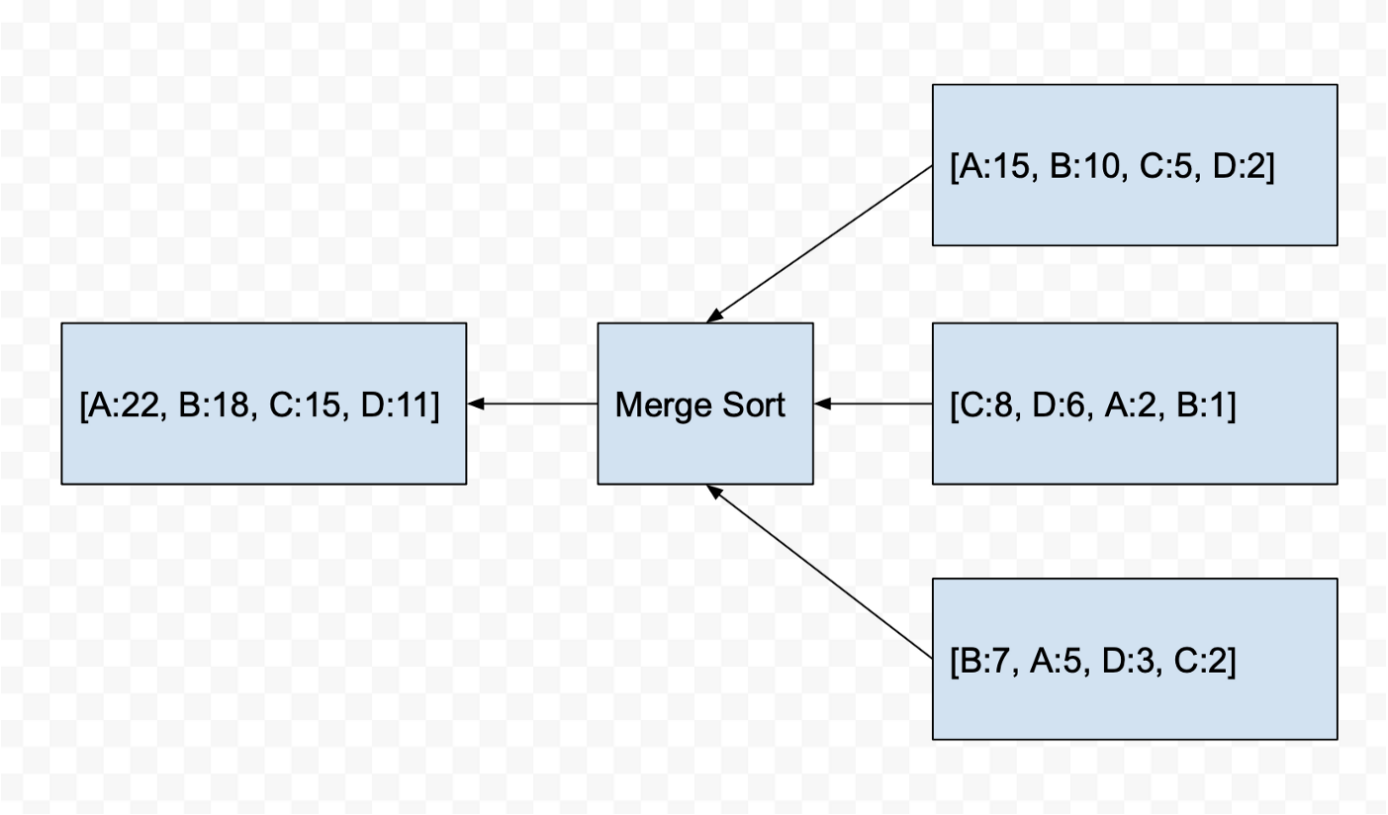
For both MySQL and Redis create a master node and 2 replica nodes. All writes go to the master and it is asynchronously replicated to the replica nodes. All reads happen from the read replicas.

Computing Top K hashtags by frequencies

In the Redis cache where we are storing the frequencies, we also maintain a **Min-Heap priority queue of size K** i.e. it stores the K most frequent hashtags.

The heap can be updated with streaming data in O(logK) time complexity.

> *We will have 12 Redis instances each storing top K frequent hashtags in the last 24 hours. In order to combine the results into a single stream of top K hashtags, we can use merge sort technique or another heap of size K that receives streaming data from each individual heap of size K.*



What if this node goes down or there is too much load on this node ?

Use multiple such instances each making requests to the 12 Redis instances whenever a GET request comes. These instances are behind a load balancer.

Count Min Sketch

With too many read requests or sudden burst in traffic, either we would be fetching stale data (possibly incorrect) for trending tags or the system would be slowed down to fetch the real time frequencies for hashtags.

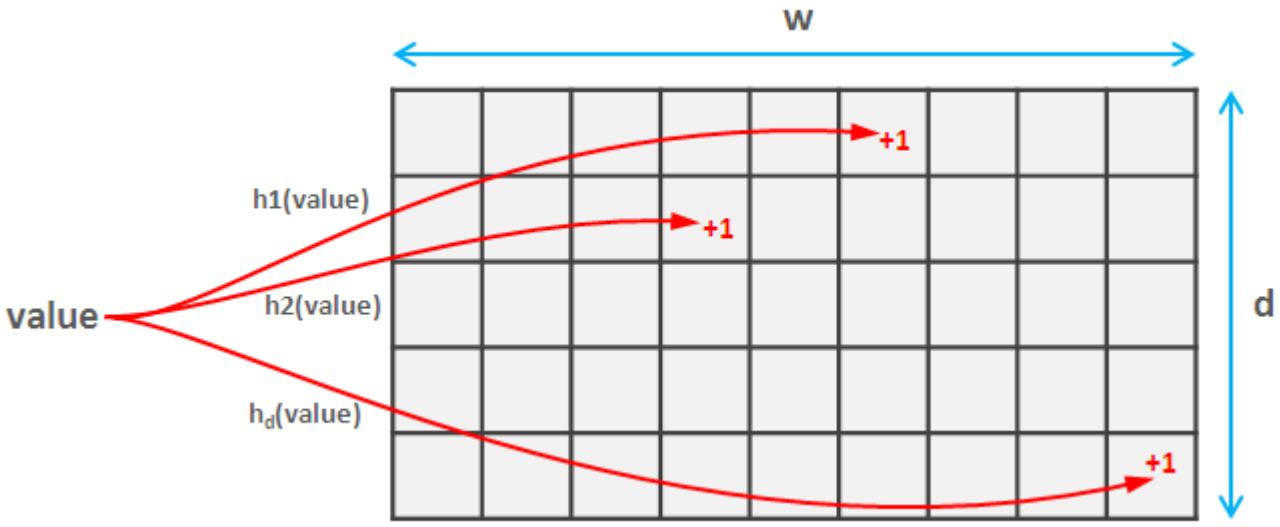
Instead of storing the frequencies of the hashtags in the last 24 hours in Hash-Map in Redis which requires 180GB of in-memory data, we can leverage something known as the Count Min Sketch data structure.

> *Count Min Sketch is a probabilistic data structure meaning that the frequencies computed may not be accurate but will have some bounded errors.*

For each hour we maintain a separate Count Min Sketch.

In each CMS, we have d independent hash functions having the range [0, w-1] for some value w. Whenever an hashtag comes in, we compute the D different hash

values H1, H2, … Hd corresponding to the d hash functions and then for each hash function we increment the cell in that row corresponding to H1, H2, … Hd by 1.



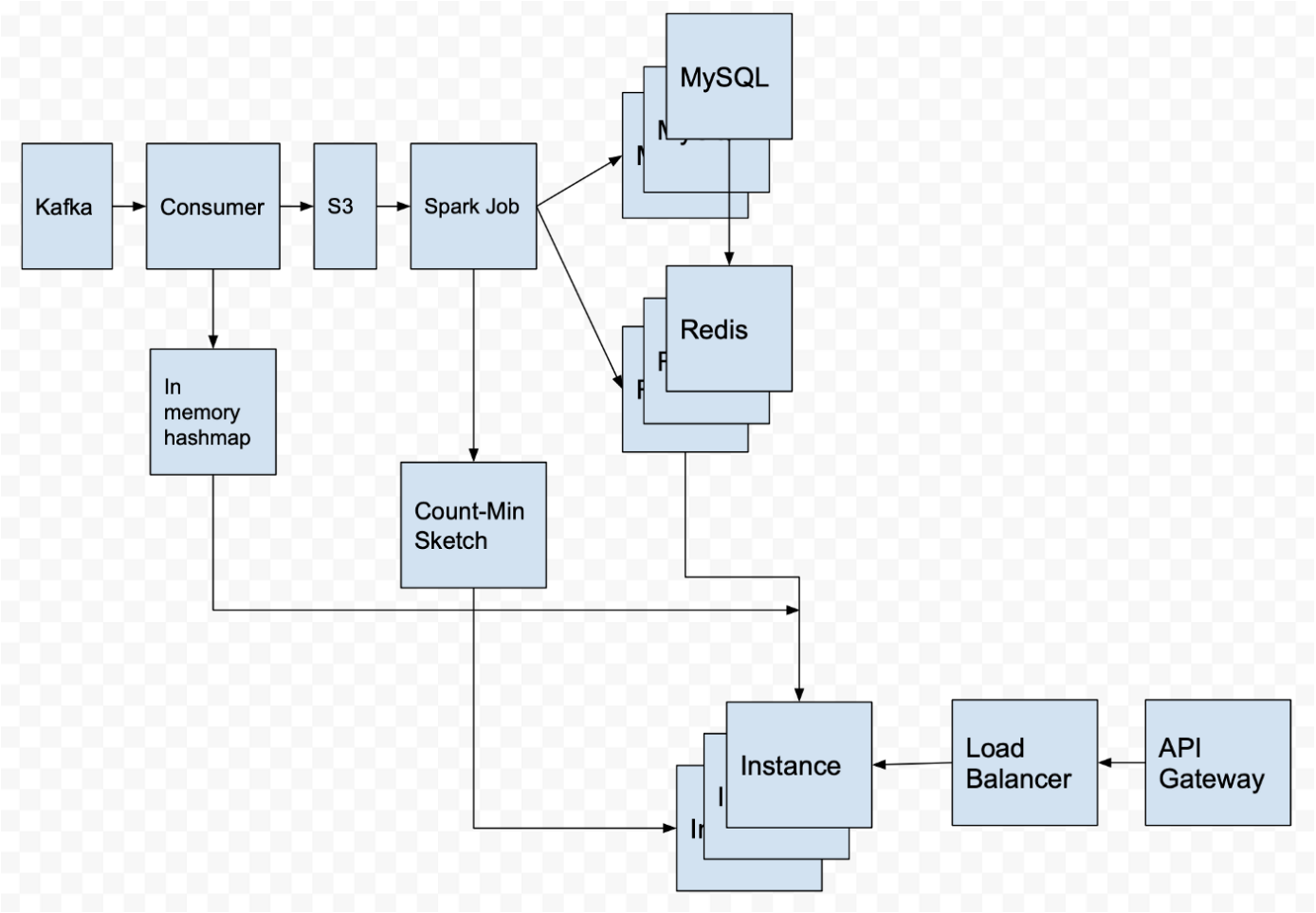
To get the frequency of a hashtag in 1 hour, again apply the d hash functions and retrieve the values from the cells H1, H2, … Hd. Take the minimum of these values as the frequency.

> *With d=10 hash functions and w=2000, we can obtain an error rate of only 0.1% (sum of all frequencies) with a confidence bound of 99.9%. Assuming each counter is of 32 bits, total size of CMS for 1 hour is 32bits*10*2000=80KB.*

For 24 hours, we will have 24 such CMS totalling 2MB only instead of 180GB.

We can have 2 pipelines — one fast and approximate and one slow and accurate. When read QPS is at peak we can use the fast pipeline to get the results else we can use the slow pipeline when traffic is low.

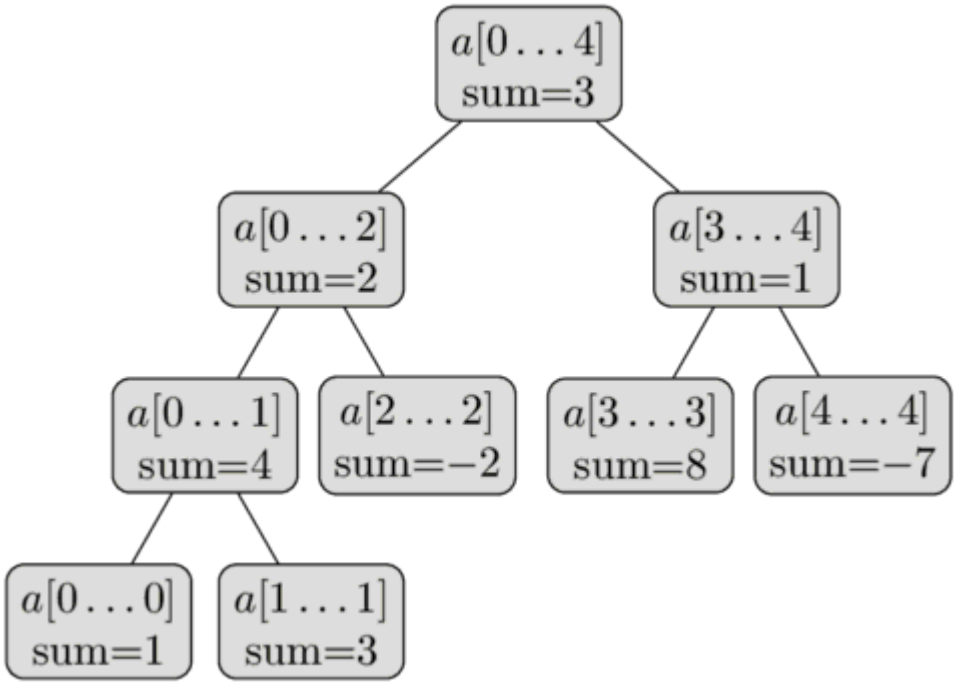
The Pipeline



Follow Up Questions

How do we handle frequently updating time interval for trending tags say 1 hour, 7 days, 30 days etc. ?

One approach is using **Segment Tree**



source: cp-algorithms

Recursively divide the number of hours for 1 year into 2 equal parts with equal number of hours. For example, for start and end times (in hours), compute mid= (start+end)/2 and create left and right nodes [start, mid] and [mid, end] respectively.

At the next level, compute [start, (start+mid)/2], [(start+mid)/2, mid], [mid, (end+mid)/2] and [(end+mid)/2, end].

Stop splitting nodes when [start, end] is less than equal to 1 hour.

We have already computed the Count-Min Sketches for each hour earlier. For a year there will be 8760 hours i.e. 8760 count min sketches each of size 80KB, thus total size is 685MB.

Each non-leaf node stores merged CMS which is the sum of all counters from individual CMS in the subtree.

The number of CMS in the whole tree is N*logN = 115K.

Thus total size of the segment tree = 115K*80KB = 9GB. Thus we can store the entire segment tree of CMS in a single Redis instance.

For a query for fetching the frequencies between time intervals [left, right], go to all the non-leaf and leaf nodes in the Segment Tree that covers the interval and add up the frequencies for all these nodes.

Useful References

- http://dimacs.rutgers.edu/~graham/pubs/papers/cmsoft.pdf
- https://barnasahadotcom.files.wordpress.com/2016/01/lec3-haritha-1.pdf
- https://cp-algorithms.com/data_structures/segment_tree.html
- https://leetcode.com/discuss/interview-question/system-design/243604/Design-a-real-time-dashboard-showing-the-most-played-songs