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

***Problem Statement*** : Identify the top K trending events, like popular hashtags on LinkedIn, most liked photos on Instagram, or most played songs on Spotify within specified time frames such as the last 1 minute, 5 minutes, 1 hour, and 1 day.

***Functional Requirement***: Find top K heavy hitters/events in last 1 minute, 5 minutes, 1 hour or 1 day.

***Non-Functional Requirements*** :

* Highly Scalable : The system should be capable of handling a massive amount of events, up to billions of events per day.
* Highly Available: The system should maintain high availability to ensure uninterrupted access to trending event data.
* Performance : The system should have a low latency. It should return top K list in a few milliseconds.

How to approach this problem?

The naïve solution

***Solution 1*** : Hashmap and Heap Based Approach (Single Server, In-Memory):

We will create a hashmap to store occurrence of incoming event stream and min heap of size K to store top K event list for each time frame.

Whenever an event occurs within a specific time frame, its frequency will be updated in the hashmap, and if this updated frequency impacts the top K events, the min heap will also be updated accordingly.

At the end of each time window, the results can be dumped to a database, and the algorithm can restart for the next window.

A diagram of a block diagram

Description automatically generated

Hashmap & Heap Single Server

*Pros*: It offers low latency due to the utilisation of an in-memory hash-map.

*Cons*: Scalability is a concern as maintaining all event occurrences in a single server’s hash-map becomes impractical at the scale of billions of events per day.

Solution 2

**Multiple server hashmap and heap-based approach**

***Solution 2*** : Horizontal Scaling with Multiple Server Hashmap and Heap-Based Approach:

Given the massive number of events, storing them on a single server hash -map becomes impractical. To address this, we horizontally scale the initial solution.

Here, events are partitioned across servers using shard key, ensuring each server manages a unique subset of events. Each server maintains its own hash-map and min heap.

At the conclusion of each time window, all servers provide their top K event lists, which are then merged using a merging service to produce the final top K list.

This is very similar to merging k sorted lists in system design

The final result is then stored in the database.

A diagram of a diagram

Description automatically generated

Horizontally scaling initial solution

*Problem with this solution ?*

Even though we have horizontally scaled, the problem of scalability remains. What if the number of events keep increasing?

Now we will solve this problem in 2 parts.

***Solution 3 (Part 1)*** : — Count Min Sketch and Min Heap (Single Server, In-Memory)

And below is when count min sketch is a good choice here

We have observed that the Count-min sketch is a good choice in a situation where we have to **process a large data set with low memory consumption**.

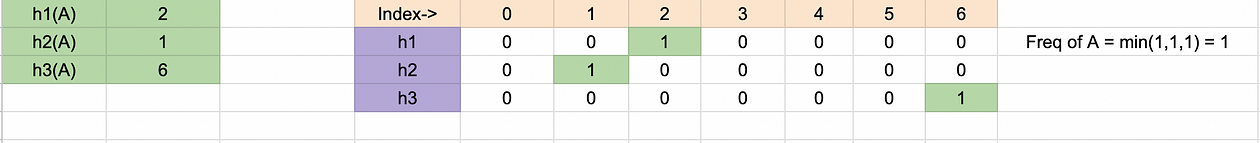
As the number of events continue to rise, horizontal scaling becomes expensive. To tackle this, we will use count min sketch which is a probabilistic data structure that serves as a frequency table of events in a stream of data.

Count Min Sketch has a fixed size and utilises hash functions to assign events their frequencies. Unlike a hash table, it takes up less space but may overcount some events due to collisions.

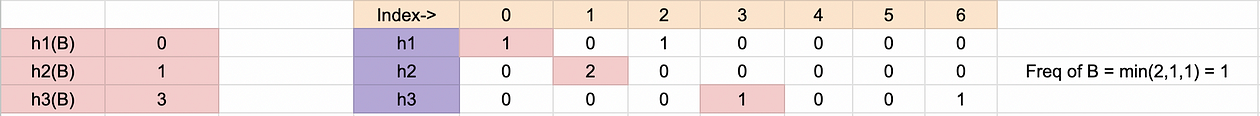
Suppose we have a Count Min Sketch of size 3\* 7, representing 3 different hash functions.

1. For each event, all the 3 hashes are calculated, and each hash-value is then modulo by 7.
2. Subsequently, the corresponding cell for each generated hash-value is incremented by 1.
3. Then the smallest frequency among these 3 cells is considered as the event’s frequency (hence the “min” in Count Min Sketch). If necessary, the min heap is updated accordingly.

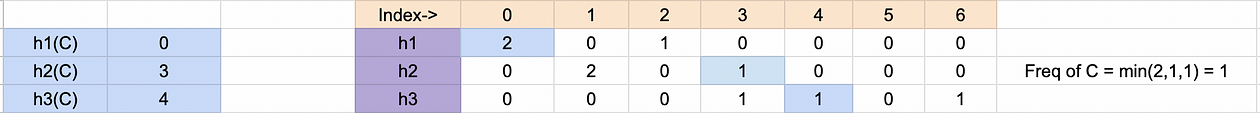
Understanding Count min sketch by an example:



On occurrence of event A



On occurrence of event B



On occurrence of event C

At the end of each time window, the results are flushed to the database, and the Count Min Sketch matrix is reset to 0.

A diagram of a diagram

Description automatically generated

Count Min Sketch & Min Heap Single Server Solution

Since the size of a Count Min Sketch is independent of the number of events, it serves as a scalable alternative to a hash-map. This addresses the challenge of horizontal scaling in the initial solution. However, it provides approximate results.

*Pros* : It is real time and fast. Count min sketch is of fixed size. It is simple and cost effective.

*Cons* : Results are approximate.

***Solution 3 (Part 2)*** : — File Storage and Map Reduce Job :

This solution aims to provide precise results. We will use a distributed file storage system, such as HDFS, to store event logs. These logs will be consumed via a Kafka queue and then saved in HDFS.

What’s map reduce?

1. Please see below

To process these events, we will run two MapReduce jobs:

1. The first job will calculate the frequency of each event within a specific time frame, aggregating the event counts.
2. The second job will determine the top K events from these aggregated counts.

We can run these MapReduce jobs on an hourly basis. However, this approach might take some time to perform the calculations to determine top K events for specified time window. The resulting data will be stored in a database.

A diagram of a computer process

Description automatically generated

HDFS and Map Reduce Job Solution

*Pros*: Provides precise results.

*Cons*: Not real-time.

***Solution 4*** : Final Combined Solution

We can combine both parts of Solution-3 to achieve our objectives. Solution-3 (Part 1) provides real-time and approximate results, while Part 2 offers precise but non real-time results.

**Common Components for both the solutions :**

* API gateway : We’ll make use of the API gateway to redirect any incoming events to our backend service.
* Kafka : We will utilise a messaging queue such as Kafka to process events for both solutions effectively.
* Database : We can utilise any SQL/ No SQL database to store the results.

**Steps for final solution :**

1. Event streams will arrive at the API Gateway.
2. The API Gateway will forward all events to Kafka.
3. Two paths will consume events from Kafka:

* Path 1: A single server using Count Min Sketch to calculate the top K list in real-time and save it to the database.
* Path 2: An HDFS consumer will save the event data on HDFS. A MapReduce job will then aggregate event frequencies over a given time window. Another MapReduce job will determine the top K events.   
    
  Finally, the precise top K list is saved to the database.

A diagram of a diagram

Description automatically generated

Final Solution

Given the significant volume of events, further optimisations include:

1. Performing aggregation of events at the API gateway level before forwarding them to the Kafka queue.
2. Using a Data Partitioner to read batches of events from Kafka and partition them into individual events. The partitioner will use a key derived from the event and time window to hash the events into different partitions. These partitioned events will then be sent to another Kafka queue. A Partition Processor will handle all events and write them in the same order.

If you found the post helpful, don’t forget to shower us with as many claps :)

How does count min sketch work in code?

*Count min sketch and is defined as a simple technique to summarize large amounts of frequency data. Count-min sketch algorithm talks about keeping track of the count of things. i.e, How many times an element is present in the set.*

**What is Count-Min Sketch?**

Count-min sketch approach was proposed by **Graham Cormode** and **S. Muthukrishnan**. in the paper [**approximating data with the count-min sketch**](https://ieeexplore.ieee.org/document/6042851) published in 2011/12. Count-min sketch is used to count the frequency of the events on the streaming data. Like the **Bloom filter**, Count-min sketch algorithm also works with hash codes. It uses multiple hash functions to map these frequencies on to the matrix (Consider **sketch** here a [two dimensional array](https://www.geeksforgeeks.org/multidimensional-arrays-in-java/) or [matrix](https://www.geeksforgeeks.org/matrix/)).

**Need for Count-Min Sketch**

Since Count-Min Sketch is used to find the frequency of an element, one might think if there is actually a need for such data structure! The answer is Yes. Let us see with the help of an example.

Let us try to solve this frequency estimation problem using a **MultiSet Data Structure**

**Trying MultiSet as an alternative to Count-min sketch**

Let’s try to implement this data structure using [MultiSet](https://www.geeksforgeeks.org/multiset-interface-guava-java/) with the below source code and try to find out the issues with this approach.

* Java

*We can easily understand that* ***as data grows, the above approach is consuming a lot of memory and time to process the data.***

This can be optimised if we use the **Count-Min Sketch**.

**How does Count-Min Sketch work?**

Let’s look at the below example step by step.

**Creating a Count-Min Sketch using Matrix**

* Consider the below 2D array with 4 rows and 16 columns, also the number of rows is equal to the number of hash functions. That means we are taking four hash functions for our example. Initialize/mark each cell in the matrix with zero.

***Note:*** *The more accurate result you want, the more hash function to be used.*

A number grid with green numbers

Description automatically generated with medium confidence

**Now let’s add some elements (Input) to the Count-Min Sketch.**

To do so we have to pass that element with all four hash functions which will result as follows.

* **Input 1: 192.170.0.1**

*Passing the input through Hash Functions:*

* ***hashFunction1(192.170.0.1): 1***
* ***hashFunction2(192.170.0.1): 6***
* ***hashFunction3(192.170.0.1): 3***
* ***hashFunction4(192.170.0.1): 1***

*Now visit the indexes retrieved above by all four hash functions and mark them as 1.*

*A screenshot of a computer code

Description automatically generated*

* **Input 2: 75.245.10.1**

*Passing the input through Hash Functions:*

* ***hashFunction1(75.245.10.1): 1***
* ***hashFunction2(75.245.10.1): 2***
* ***hashFunction3(75.245.10.1): 4***
* ***hashFunction4(75.245.10.1): 6***

*Now visit the indexes retrieved above by all four hash functions and mark them as 1.*

*Now, take these indexes and visit the matrix, if the given index has already been marked as 1. This is called* [***collision,***](https://www.geeksforgeeks.org/collision-resolution-techniques/)*i.e., the index of that row was already marked by some previous inputs.*

*In this case, just increment the index value by 1.*

*In our case, since we have already marked index 1 of row 1 i.e., hashFunction1() as 1 by previous input, so this time it will be incremented by 1, and now this cell entry will be 2, but for the rest of the index of rest rows, it will be 0, since there was no collision.*

*A screenshot of a computer code

Description automatically generated*

* **Input 3: 10.125.22.20**

*Passing the input through Hash Functions:*

* ***hashFunction1(10.125.22.20): 3***
* ***hashFunction2(10.125.22.20): 4***
* ***hashFunction3(10.125.22.20): 1***
* ***hashFunction4(10.125.22.20): 6***

*Lets, represent it on matrix, do remember to increment the count by 1 if already some entry exist.*

*A screenshot of a computer code

Description automatically generated*

* **Input 4: 192.170.0.1**

*Passing the input through Hash Functions:*

* ***hashFunction1(192.170.0.1): 1***
* ***hashFunction2(192.170.0.1): 6***
* ***hashFunction3(192.170.0.1): 3***
* ***hashFunction4(192.170.0.1): 1***

*Lets, represent it on matrix, do remember to increment the count by 1 if already some entry exist.*

*A screenshot of a number code

Description automatically generated with medium confidence*

**Testing Count-Min Sketch data structure against Test cases:**

Now let’s test some element and check how many time are they present.

* **Input 1: 192.170.0.1**

*Pass above input to all four hash functions, and take the index numbers generated by hash functions.*

* ***hashFunction1(192.170.0.1): 1***
* ***hashFunction2(192.170.0.1): 6***
* ***hashFunction3(192.170.0.1): 3***
* ***hashFunction4(192.170.0.1): 1***

*Now visit to each index and take note down the entry present on that index.*

*A screenshot of a computer code

Description automatically generated*

*So the final entry on each index was* ***3, 2, 2, 2****.*

*N ow take the minimum count among these entries and that is the result. So min(3, 2, 2, 2) is 2, that means the above test input is processed 2 times in the above list.*

*Hence* ***Output (Frequency of 192.170.0.1) = 2.***

* **Input 2: 10.125.22.20**

*Pass above input to all four hash functions, and take the index numbers generated by hash functions.*

* ***hashFunction1(10.125.22.20): 3***
* ***hashFunction2(10.125.22.20): 4***
* ***hashFunction3(10.125.22.20): 1***
* ***hashFunction4(10.125.22.20): 6***

*Now visit to each index and take note down the entry present on that index.*

*A screenshot of a computer code

Description automatically generated*

*So the final entry on each index was* ***1, 1, 1, 2****.*

*Now take the minimum count among these entries and that is the result. So min(1, 1, 1, 2) is 1, that means the above test input is processed only once in the above list.*

*Hence* ***Output (Frequency of 10.125.22.20) = 1.***

**I**

A screenshot of a computer

Description automatically generated

**Time and Space Complexity of Count-Min Sketch Data Structure**

Now let’s look at the time and space consumed with this type of approach (wrt to above Java-Guava Implementation)

| **Number of UUIDs** | **Multiset Insertion Time(ms)** | **CMS Insertion Time(ms)** |
| --- | --- | --- |
| 10 | <25 | 35 |
| 100 | <25 | 30 |
| 1,000 | 30 | 69 |
| 10,000 | 257 | 246 |
| 100,000 | 1200 | 970 |
| 1,000,000 | 4244 | 4419 |

Let’s have a look at the memory (space) consumed:

| **Number of UUIDs** | **Multiset JVM heap used(MB)** | **CMS JVM heap used(MB)** |
| --- | --- | --- |
| 10 | <2 | N/A |
| 100 | <2 | N/A |
| 1,000 | 3 | N/A |
| 10,000 | 9 | N/A |
| 100,000 | 39 | N/A |
| 1,000,000 | 234 | N/A |

**Applications of Count-min sketch:**

* Compressed Sensing
* Networking
* NLP
* Stream Processing
* Frequency tracking
* Extension: Heavy-hitters
* Extension: Range-query

**Issue with Count-min sketch and its solution:**

What if one or more elements got the same hash values and then they all incremented. So, in that case, the value would have been increased because of the [hash collision](https://www.geeksforgeeks.org/hashing-set-1-introduction/). Thus sometimes (in very rare cases) Count-min sketch overcounts the frequencies because of the hash functions.

So the more hash function we take there will be less collision. **The fewer hash functions we take there will be a high probability of collision**. Hence it always recommended taking more number of hash functions.

**Conclusion:**

We have observed that the Count-min sketch is a good choice in a situation where we have to **process a large data set with low memory consumption**. We also saw that the more accurate result we want the number of hash functions(rows/width) has to be increased.

Join [GfG 160](https://www.geeksforgeeks.org/courses/gfg-160-series?utm_source=geeksforgeeks&utm_medium=bottomtext_default&utm_campaign=premium), a 160-day journey of coding challenges aimed at sharpening your skills. Each day, solve a handpicked problem, dive into detailed solutions through articles and videos, and enhance your preparation for any interview—all for free! Plus, win exciting GfG goodies along the way! - [Explore Now](https://www.geeksforgeeks.org/courses/gfg-160-series?utm_source=geeksforgeeks&utm_medium=bottomtext_default&utm_campaign=premium)

What’s map reduce here?