**Caffeine cache**

Main difference between Cache and map is.. map don’t have eviction policy.

Hit Rate:

Used to find performance of Cache.

For example, if a cache is accessed 100 times and 85 of those accesses result in hits, the hit rate would be 85%.

When will Hit rate will be lower:

Lets say you have FIFO eviction policy,

When the cache is full, it will remove the first item and adds the new item at last. But In some case first item can be frequently accessed item, so when a person ask for that data(which is previously first) cache doesn’t have it. Now it’s a miss. If there is a miss then Hit rate will be less

Hit rate of Map:

A **map** does not have a fixed size limit and can dynamically adjust its capacity, so it doesn't evict items. So hit rate will be high than cache but you need to compromise of speed and memory

How Caffeine cache have optimal hit rate?

Using TinyLFU (Tiny least frequently used)]

TinyLFU

TinyLFU is a caching algorithm designed to manage memory efficiently while maintaining a balance between space complexity and hit rate. Also unlike traditional LFU it won’t track the frequency.

Key Concepts of TinyLFU

**Two-Queue Structure**:

Main cache

Frequency counter

**Probabilistic Frequency Tracking**:

It keeps track of items that have been accessed recently and assigns a "frequency" score based on their access patterns.

**Eviction Policy**:

TinyLFU evicts the item that has the lowest access frequency score.

**Window Mechanism**:

TinyLFU uses a sliding window approach to limit how far back it tracks item usage.

**Internal working:**

**Count-Min Sketch Structure**

Count-Min Sketch is a method used to estimate the frequency of items in a data stream while using minimal memory.

Whenever you search an input, multiple hash values are found using multiple hash function. In array of Hash Algorithm (y axis) and Hash value (x axis) add the counter for each inputs hash value.

**Eg:**

input - Eswar

hash algorithm- H1, H2, H3, H4

Hash value obtained when applying each hash function on input- 0, 1, 3, 2

You increment the counters at the positions specified by the above hash values for "Eswar."

Array output for Eswar:

**0 1 2 3 4**

**H1** *1 0 0 0 0*

**H2** *0 1 0 0 0*

**H3** *0 0 0 1 0*

**H4** *0 0 1 0 0*

*Increase the count while finding the hash for other inputs*

Find frequency of a input:

Firstly find Hash value using same hash function for an input. Now find the count from diff position of the array based on got found hash value. The Minimum Count value there is the frequency of the input.

Eg array:

Find output for Eswar:

**0 1 2 3 4**

**H1** *1 0 0 0 3*

**H2** *0 2 0 0 1*

**H3** *5 0 0 3 0*

**H4** *2 0 2 0 0*

Hash value obtained when applying each hash function (H1,H2,H3,H4) on input is 0, 1, 3, 2

Minimum value while searching about found index is 1 (H1,0)

A diagram of a program

Description automatically generated

**Space Efficiency**

This counter value is 4 bit value (0-15) instead of integer (-2^31 to 2^31)

**Doorkeeper Optimization (**Bloom filter)

A Bloom filter is a probabilistic data structure used to test whether an element is a member of a set.

The Bloom filter is designed to quickly tell whether an item has been seen before, using a bit array and hash functions. It uses less memory compared to the CMS and provides a probabilistic answer (with a possibility of false positives).

* Before an item's frequency is tracked in Count-Min Sketch, The Bloom filter acts as a first-time visitor filter which is used to find the item is previously accessed or not.
* This avoid the CMS to add counter for onetime accessible items.
* Only items that pass through the Bloom filter (seen before) get counted in Count-Min Sketch

It is a probable algorithm

* + If it says item is not present then for sure searched item is not present in the list.
  + If it says item is present then for searched item might be present in the list(probable).

A screenshot of a computer flowchart

Description automatically generated

Internal Working:

It internally uses BitSet.

It is a dynamic array, Each position in BitSet indicates whether it's occupied (true) or not (false)

Eg:

*BitSet bit = new BitSet(bit\_size);*

*bit.set(1);*

*bit.get(1); //true*

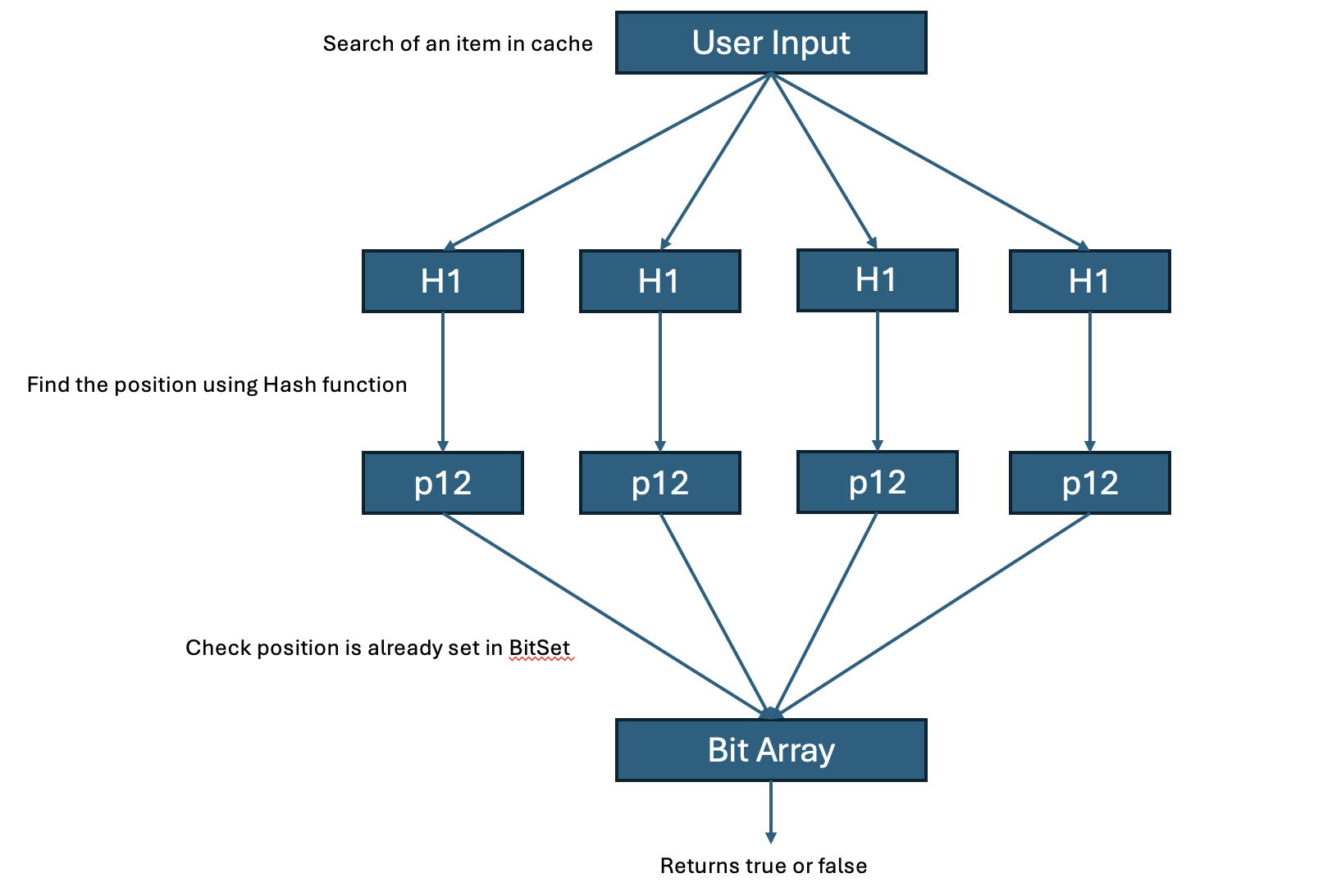
This BitSet should be created with an initial size. This initial size is calculated based on a formulae

Bit size is calculated using two inputs:

* Cache size
* Desired false positive rate

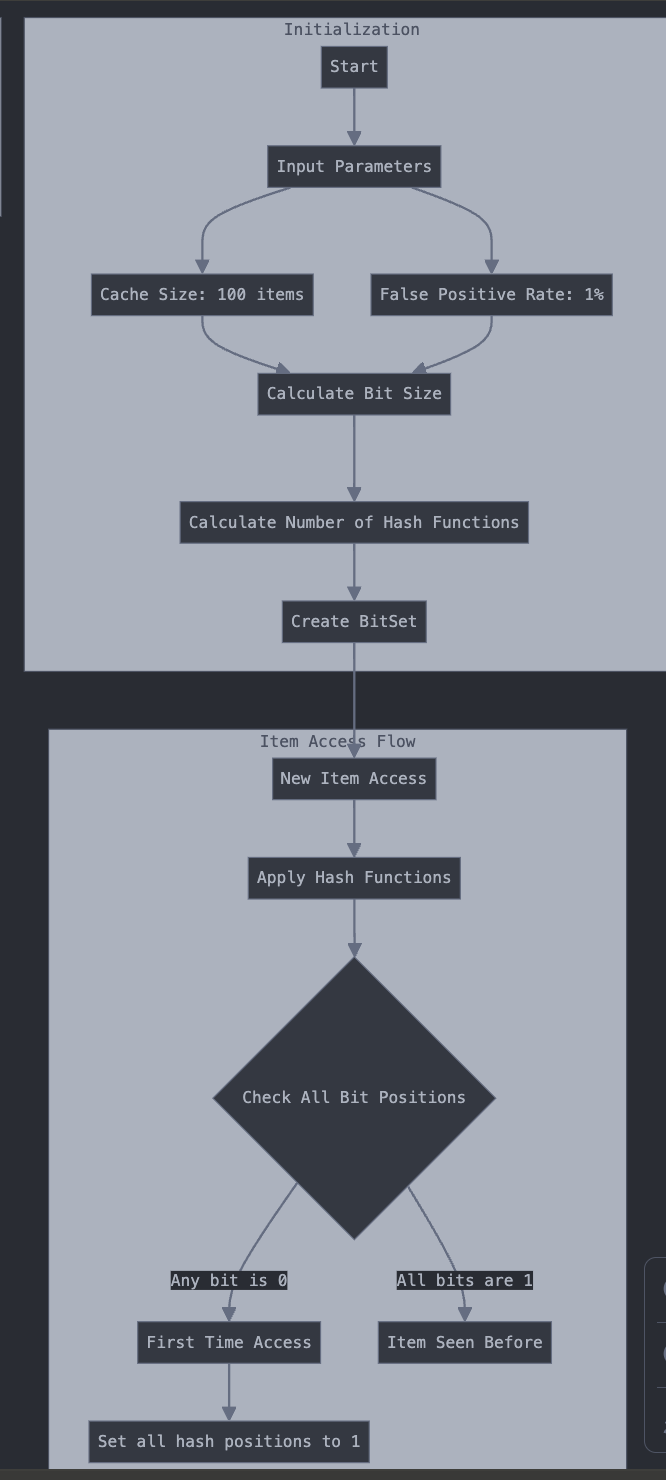
Lets say I have cache size of 100 and need to have false positive rate as 1% (means 99% correct when it says value is already accessed), using this I will find bit size.

For example bit size we got bit size as 400. So Number of hash functions = bit size / cache size (e.g., 400/100 = 4 hash functions)



When accessing an item:

* Apply all hash functions to get multiple positions
* Check if ALL those positions in BitSet are set
* If all positions are set → item was likely accessed before
* If any position is not set → definitely first access



**Aging Mechanism:**

Purpose of Aging:

Items that were once popular but are no longer frequently accessed would retain high counts, potentially blocking newer, more relevant items from entering the cache. So it prevent older and less relevant items from dominating the frequency counts.

Working:

a. Periodic Reset: At regular intervals or when certain conditions are met (e.g., after a specific number of requests), the frequency counters for all items are reduced.

b. Halving Counts: The most common implementation of aging in tinyLFU involves dividing all frequency counts by 2. This operation is also known as a "reset".

So before sending item to CMS, the data will be inserted in sliding window. Once the window reaches it limit, the first element will removed. If that removed element is not present in the window then we can make halving count or reset.

A diagram of a window

Description automatically generated

Eg:

**Initial Setup**

* **Cache Size**: 3
* **Access Pattern**: A, A, B, A, C, D, B, C, B, D, C
* **Sliding Window Size**: 5 (for demonstration)
* **CMS**: Initially empty.
* **Sliding Window**: Initially empty.

**Step-by-Step Breakdown**

1. **Access A**:
   * **Action**: Add A to the cache and increment its count in the CMS.
   * **Cache**: [A]
   * **CMS**: {A: 1}
   * **Sliding Window**: [A]
2. **Access A** (again):
   * **Action**: Increment the count for A.
   * **Cache**: [A]
   * **CMS**: {A: 2}
   * **Sliding Window**: [A, A]
3. **Access B**:
   * **Action**: Add B to the cache.
   * **Cache**: [A, B]
   * **CMS**: {A: 2, B: 1}
   * **Sliding Window**: [A, A, B]
4. **Access A** (again):
   * **Action**: Increment the count for A.
   * **Cache**: [A, B]
   * **CMS**: {A: 3, B: 1}
   * **Sliding Window**: [A, A, B, A]
5. **Access C**:
   * **Action**: Add C to the cache and increment its count in the CMS.
   * **Cache**: [A, B, C]
   * **CMS**: {A: 3, B: 1, C: 1}
   * **Sliding Window**: [A, A, B, A, C]
6. **Access D**:
   * **Action**: The cache is full. Evict C (lowest frequency among the remaining items).
   * **Cache**: [A, B, D]
   * **CMS**: {A: 3, B: 1, C: 1, D: 1}
   * **Sliding Window**: [A, A, B, A, C, D]
     + Sliding Window size limit reaches so first element ‘a’ is removed, but is present in last 5 access. So reset is not done for ‘a’
7. **Access B** (again):
   * **Action**: Add B to the cache.
   * **Cache**: [A, D, B]
   * **CMS**: {A: 3, B: 2, C: 1, D: 1}
   * **Sliding Window**: [A, B, A, C, D, B]
     + Sliding Window size limit reaches so first elemen