Garrett Kemper

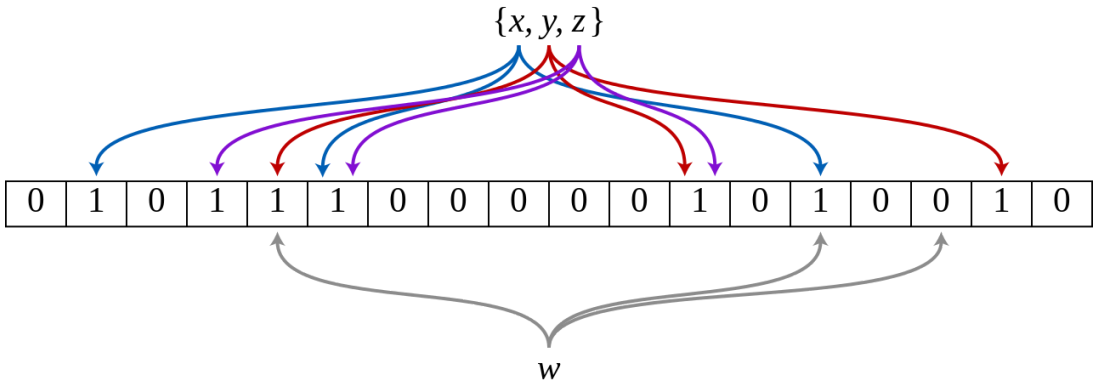
CSC 212 Research Project

04/11/24

Bloom Filter

The Bloom Filter is a probability based data structure created by Burton Howard Bloom in 1970. The main functionality of this structure is to quickly test whether an element is in a set. Since being probabilistic, it does have the possibility of returning false positives. However, a false negative is impossible for this structure. Thus, the output of a query in a Bloom Filter is only “possibly in set” or “definitely not in set.”

A Bloom Filter consists of a bit array of length m. An empty bloom filter has a bit array of all zeros. To add an element, the element is hashed k times and the indices found through those hashes are all set to 1. To query an element, the same hash functions are used to check whether all of the indices are set to 1. If any of the indices are 0, then the element is definitely not in the set as adding it would have made those indices all 1. If all of the indices are 1, then it is possible that the element is in the set. There is a possibility that the addition of other elements may have caused those indices to be set to 1 by coincidence, hence why it is only able to say it is possibly in the set.



https://doris.apache.org/docs/1.2/data-table/index/bloomfilter/

The advantages of this structure is that it is highly space and time efficient. Using a Bloom Filter can greatly reduce the space needed for a large set as it does not actually store the elements in it. Additionally, insertions and queries can be very quick compared to a large set as it does not need to make any comparisons to other elements and only needs to compute the hash functions. As a result, Bloom Filters have a time complexity of O(k) and a space complexity of O(m), where k is the number of hash functions and m is the number of bits in the bit array. This makes Bloom Filters much more efficient than other structures such as binary trees and searching algorithms.

The main disadvantage of this structure is that it is possible to return false positives. Some level of accuracy has to be sacrificed in order to achieve higher efficiency for this structure. However, it is possible to minimize the chances of this. The probability of a false positive can be calculated using the equation $P = (1 - (1 - \frac{1}{m})^{kn})^{k}$ where P is the probability, m is the length of the bit array, k is the number of hash functions, and n is the expected number of elements. The ideal bit array length can be calculated using the equation $m = -\frac{n\,ln(P)}{(ln2)^2}$ and the ideal number of hash functions using $k = \frac{m}{n}ln(2)$ where P is your desired probability. Another disadvantage to Bloom Filters is that elements cannot be removed from the set. If needed, there are alternate variations of Bloom Filters that can allow for the removal of elements.

The Bloom Filter was chosen for this project due to its fast query speeds and space efficiency. This project is working with large amounts of data and having speed and efficiency is crucial in order for the project to be successful. Bloom Filters were utilized in order to store the classes that each student has already passed. These were larger sets and would have taken up significantly more space if a set was used. The program also makes many checks with these

classes throughout, so a fast query is needed in order to make this process quick. Additionally, the false positives are not a significant concern as the main goal of these queries is to check that a class has not been taken already. It also was easier to mitigate the false positives since the possible entries to the Bloom Filters are known and the hash functions can be designed in a way that mitigates any overlap between entries.

Notes:

- Probability based data structure
- Burton Howard Bloom in 1970
- Test whether an element is in a set
- Can have false positives but not false negatives.
    - Returns “possibly in set” or “definitely not in set”
- Good for large amounts of source data
    - Lower memory usage compared to error-free hashing
- <10 bits per element required for a 1% false positive probability
    - Regardless of size or number of elements in the set

Functionality:

- Bit array of m bits all set to 0
- k defined hash functions
    - Hashes elements to one of m positions
- k is small constant, m is proportional to k and number of elements to be added
- Adding an element:
    - Hash through each of the k hash functions and set the bits at the given indices to 1
        - Does not account for collisions (if an issue, counting bloom filter variant is possible but less memory efficient)
- To query:
    - Filter through each hash function, if any of the given indices are 0 the element is definitely not in the set
    - If all are one, either element is in set or false positive