Bloom Tree Feature Mappings

James Pearce
pearja@amazon.com

June 16, 2019

Abstract

We introduce a new memory efficient data structure appropriate for many to one mappings. This new data structure, which we refer to as a Bloom Tree Map, is built from a combination of a Bloom filter and Binary Search Tree with an optional Hash Map. We show that in some situation such a data structure can offer a 10× increase in capacity for the same memory as a standard Hash Map.

Related work

Bloom filters

A Bloom filter is a memory-efficient data structure used to test element membership in a set. Bloom filters are probabilistic data structures since they have a non-zero false positive probability for any given element in a set membership test. However, the false negative probability, on the other hand, is always zero by design.

For example, a Bloom filter, BF, constructed with the set $S = \{x_1, x_2, ..., x_n\}$ will evaluate to "True" for test set element x^* if $x^* \in S$ always. However if $x^* \notin S$ then BF may evaluate to "True" with a false positive probability θ. In otherwords

$$\theta = P(BF(x^*) = \text{True} | x^* \notin S). \tag{1}$$

Bloom filters use a set of hash functions, $H = \{h_1, h_2, ..., h_j\}$, to encode set membership in a bit array. A bloom filter is constructed by setting all bits $H(x^*)$ (H should be considered a multivalued function of x^*) to one in a

empty bit array b. Thus for element x^* we have $b[h_j(x^*)] = 1$. This process of setting bits is repeated for every element in S. Set membership is then evaluated for a given element by checking each of the positions $H(x^*)$ in b. If all positions equal one then $BF(x^*)$ returns "True" else it returns "False".

False positives occur when there are hash collisions such as $H(x_i) = H(x_j)$ where $x_i \in S \wedge x_j \notin S$.

As shown by [**?**] the false positive probability θ can be calculated with m the total number of bits in b, n the cardinality of S and k the cardinality of H:

$$\theta = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \tag{2}$$

This result follows from the "Birthday problem" and the definition of e in its limit form. We will denote the number of hash functions that minimizes θ as $\hat{k}$, which can be found to be

$$\hat{k} = \frac{m}{n} \ln 2 \tag{3}$$

By substituting $\hat{k}$ into 2 one obtains the optimal number of bits per element

$$\frac{m}{n} = -\alpha \ln \theta \tag{4}$$

where $\alpha = (\ln 2)^{-2} \approx 2.1$. This result along with 3 shows the optimal number of hash functions is determined by

$$\hat{k} = -\frac{\ln \theta}{\ln 2} = -\log_2 \theta. \tag{5}$$

This result shows that the required number of hash functions depends only on the target false positive probability. These four equations will be useful in the Bloom Tree derivations that follows.

Previous work on Bloom Trees

Bloom Tree

The many-to-few problem

A Bloom Tree, as presented here, is a new data structure designed to efficiently assign a set of keys, $\mathcal{K}$ to a set of values, $\mathcal{V}$, through a many-to-one

mapping where $|\mathcal{K}| \gg |\mathcal{V}|$. This differs from a standard hash map which assigns keys to values through a one-to-one mapping. The requirement that $|\mathcal{K}| \gg |\mathcal{V}|$ is not strictly necessary, however, as we'll see, most of the memory saving of the Bloom Tree over a hash map come from this cardinality imbalance between $\mathcal{K}$ and $\mathcal{V}$. For this reason we refer to the ideal use case as the *many-to-few problem*. [Examples of many to few]

Bloom Tree miss-classification probability

The basic idea of a Bloom Tree is to use a binary search tree (BST) to assign the keys $\mathcal{K}$ to their respective values $\mathcal{V}$, which are encoded in the terminal nodes of the BST. All internal nodes of the BST are associated with a Bloom filter that checks for set membership of a given key in the right child node (right node instead of left is an arbitrary choice). If the the BF returns "True" the right child node is chosen, otherwise the key defaults to the left node. By repeating this process the Bloom Tree is traversed from the root node to one of the terminal nodes. The final terminal node value is taken to be the associated value to the key.

As discussed in the previous section, there is a false-positive probability of θ, which is associated with each internal node. This false positive probability is related to the overall miss-classification probability of the Bloom Filter ψ_{BF} as

$$\psi_{BF} = P(x^* \notin S)P(BF(x^*) = \text{True}|x^* \notin S) = \phi\theta. \tag{6}$$

Where we have used eq. 1 and defined $P(x^* \notin S) \equiv \phi$.

The overall miss-classification probability of the Bloom Tree can derived using the same "Birthday problem" trick as in the Bloom filter false-positive probability calculation. We start by calculating the probability of true-classification, which is simply the product of true-classification probabilities of each node encountered by x^* as it descends the tree and then make repeated use of the negation rule ($\neg P \equiv 1 - P$) to calculate the miss-classification probability of the Bloom Tree ψ_{BT}:

$$\psi_{BT}(x^*) = \neg \prod_{t \in path(x^*)}^{L} \neg(\phi_t\theta_t) \tag{7}$$

where L is the number of level in the BST, $L = \log_2(|\mathcal{V}|)$ and $path(x^*)$ is the set of nodes encountered by x^* as it traverses the BST from the root to

terminal node. For now we can assume that θ_t, the false-positive probability at each node, is constant which we'll denote simply as θ as before (we can keep θ_t constant simply by increasing the bits allocated in the Bloom filter as show in eq 4). In general, however, we shouldn't expect ϕ_t to be constant, since it's value will depend on the distribution of keys over their associated values.

If p_v is the probability that our test example x^* will be associated with value v then ϕ_t can be calculated as

$$\phi_t = \frac{\sum_{l \in D_L(t)} p_l}{\sum_{l \in D_L(t)} p_l + \sum_{r \in D_R(t)} p_r} \tag{8}$$

where $D_{L/R}(t)$ is the set of terminal nodes (each terminal node is associated with a value v and probability p_v) that are left/right children descendants of the parent node t. Equally on could use the number of elements associated with each terminal node instead of p_v.

However to calculate the expected miss-classification probability, $\mathbb{E}[\psi_{BT}] = \langle \psi_{BT} \rangle$, only the expect false positive probability $\langle \phi \rangle$ is need since each ϕ_t is independent. The expected false positive probability can be calculated by averaging ϕ_t over all internal nodes, weighted by the probability of encountering node t, i.e. $P(t \in path(x^*))$. This weighting probability is given simply by the product of all ancestors split probabilities $\phi_{A(t)}$ of t, which may correspond to left (ϕ) or right ($\neg\phi$) splits. Hence

$$\langle \phi \rangle = \frac{1}{L} \sum_{t}^{|\mathcal{V}|-1} \phi_t \prod_{l \in A_L(t)} \phi_l \prod_{r \in A_R(t)} \neg\phi_r \tag{9}$$

where $A_{L/R}(t)$ is the set of all left/right ancestors nodes of t. Thus $\langle \phi \rangle$ is completely determined by the key-value distribution p_v. With this result we can derive an expression for θ, which will be useful since it determines the Bloom Filter parameters such as $\hat{k}$.

From eq 7 we have

$$\langle \psi_{\mathrm{BT}} \rangle = \neg \prod^{L} \neg(\langle \phi \rangle \, \theta) = 1 - (1 - \langle \phi \rangle \, \theta)^L. \tag{10}$$

Using the same approximation as in eq. 3 we see

$$\langle \psi_{\mathrm{BT}} \rangle \approx 1 - e^{\langle \phi \rangle \, \theta L}. \tag{11}$$

Solving for θ we finally arrive at

$$\theta = -\frac{1}{\langle\phi\rangle\, L}\ln(1-\langle\psi_{\mathrm{BT}}\rangle) \approx \frac{\langle\psi_{\mathrm{BT}}\rangle}{\langle\phi\rangle\, L} \tag{12}$$

where the approximation on the right-hand side is given by the first order Taylor expansion.

Bloom Tree allocated memory

The total allocated memory of a Bloom Tree is give simply by the aggregate of each Bloom filter's memory at each internal node in the tree. Thus the total memory of the Bloom Tree, m_{BT}, is give by

$$m_{BT} = \sum_{t}^{|\mathcal{V}|-1} m_t = -\alpha \ln\theta \sum_{t}^{|\mathcal{V}|-1} n_t^{(r)} \tag{13}$$

where $n_t^{(r)}$ is the number of elements assigned to the right splits, or number of keys in $\mathcal{K}$ encoded into the Bloom Filter associated with node t. Note that only the right assignments at each node need to be encoded into the node's Bloom filter. This is because at each node the key is being checked for set membership in the set off all keys associated with the right splitClearly $n_t^{(r)} = \neg\phi_t n_t$ where n_t is the total number of elements encoded in node t. Likewise n_t is can be calculated from the total number of keys n as $n_t = nP(t \in path(x^*))$. Hence

$$m_{BT} = -\alpha n L\langle\neg\phi\rangle \ln\theta \tag{14}$$

where we've used eq. 7 with $\langle\neg\phi\rangle$ in place of $\langle\phi\rangle$. Using eq. 12 we can calculate the total number of bits needed by the Bloom Tree in terms of the miss-classification probability as

$$m_{BT} = -\alpha n L\langle\neg\phi\rangle \ln\left(\frac{\langle\psi_{\mathrm{BT}}\rangle}{\langle\phi\rangle\, L}\right). \tag{15}$$

False-positive hash table

Having a non-zero miss-classification probability is less than desirable in a look-up table. However, since we know the total set of keys and values at the

time of BST construction we can supplement the Bloom Tree with a standard hash map (or any suitable data structure) the stores all of the false-positive examples. Then at evaluation time the hash map will be checked first, if the key is in the hash map the corresponding value is returns, otherwise the Bloom Tree is queried. This combined data structure is guaranteed to have a miss-classification rate of zero.

The memory required by a hash map is related to the size of the key, m_k, their associated values, m_v t and the total number of key-values pairs n:

$$m_{HT} \propto (m_k + m_v)n. \tag{16}$$

This is not exactly a linear relation since typically a hash map will double it's size in discrete jumps as it fills up. For example a Java hash map doubles its size when $m_{HT}/n = 0.75$.

We can write the total memory of the combined Bloom Tree hash map data structure as

$$m_{comb} = m_{BT} + \langle \psi_{\mathrm{BT}} \rangle \, m_{HT}. \tag{17}$$

where m_{HT} is taken to be the size of the hash map if filled with all key-value pairs (not just the false-positives). The optimal value of θ, which minimizes m_{comb} can be derived by finding the root of the first derivate with to $\langle \psi_{\mathrm{BT}} \rangle$:

$$\langle \hat{\psi}_{BT} \rangle = \underset{\langle \psi_{\mathrm{BT}} \rangle}{\operatorname{argmin}} \, m_{comb} = \frac{\alpha n L \langle \neg \phi \rangle}{m_{HT}}. \tag{18}$$

Plugging $\langle \hat{\psi}_{BT} \rangle$ back into eq. 17 we arrive at the result

$$m_{comb} = \alpha n L \langle \neg \phi \rangle \left[1 - \ln \left(\frac{\alpha n \langle \neg \phi \rangle}{m_{HT} \, \langle \phi \rangle} \right) \right]. \tag{19}$$

Upper bound on m_{comb}

As noted earlier, only the right splits at each node need to be encoded into the associated Bloom filter. Which means all keys assigned to left nodes do not contribute to the size of the current node's Bloom filter. In fact keys assigned to the left-most path in the Bloom Tree will never have to be encoded in the structure at all. Hence keys associated with the left-most value contribute nothing to m_{BT} (however they may contribute to m_{comb} through false-positives).

This observation leads to a natural ordering of $\mathcal{V}$ in the Bloom Tree's leaf

nodes. Values should be sorted according to key counts with values with the highest key counts (associated with the most amount of keys) placed at the left-most leaf, while those with the lows key counts placed at the right-most leaf. Ordering the leaves in such a way can significantly reduce the memory sizes m and m_{comb} for non-uniform key distributions (see results section). The worst case scenario in terms of m_{BT} is when the key count distribution over the values is uniform. In such a scenario sorting the values gives little benefit. We can use this scenario, however, to derive an upper bound on m_{BT} and m_{comb}.

In this case $\langle\phi\rangle = \langle\neg\phi\rangle = \frac{1}{2}$ exactly. Taking this result and plugging back into eq. 20 we get

$$m_{comb} = \frac{1}{2}\alpha n L \left[1 - \ln\left(\frac{\alpha n}{m_{HT}}\right)\right]. \tag{20}$$

To compare m_{comb} to it's equivalent hash table we will approximate m_{HT} as a linear function of n such as

$$m_{HT} = \beta(m_k + m_v)n \tag{21}$$

where β is a constant corresponding to the inverse of the hash map's average load factor. For Java hash maps the allocated memory doubles in size once the max load factor of 0.75 is reached, which results in a average load factor of 0.5 or $\beta = 2$. It should be noted that is linear approximation does not take into account collision handling mechanisms, which may increase memory allocation.

With is we can derive the upper bound on M_{comb} in units of m_{HT}, which we find to be

$$\frac{m_{comb}}{m_{HT}} = \frac{\alpha}{2\beta}\frac{\log_2(|\mathcal{V}|)}{m_v + m_k}\left[1 - \ln\left(\frac{\alpha}{\beta}\frac{1}{m_v + m_k}\right)\right]. \tag{22}$$

We can simplify this result by choosing some realistic values for m_v and m_k. We will be conservative and use Java long ints, $m_v = m_k = 64$(bits) (for python $m_v = m_k = 196$ is realistic). This gives us

$$\frac{m_{comb}}{m_{HT}} = 0.024\log_2(|\mathcal{V}|). \tag{23}$$

For $|\mathcal{V}| = 1024$ the combined data structure is about one quarter of a standard Java hash map. In the results section we'll show that one can do significantly better than is upper bound when the key count distribution over the values is unbalanced.

Implementation details

One Bloom filter, many hash functions

Efficient hash generation

Algoritm

Empirical results

Memory efficiency

Query latency