**Bloom Filter for Big Data File Systems**

**I. Abstract**

This project aims at implementing a file system metadata scheme that uses bloom filter to avoid unnecessary SSD/disk accesses. This mechanism is especially useful for big data file systems (i.e. HDFS, ClusterFS) where at least hundreds of thousands of files exist in each directory and analytical queries over those files are required. The experiment shows that our method dramatically improved file query performance compared to the existing scheme. In addition, the implementation overhead of the proposed scheme is small.

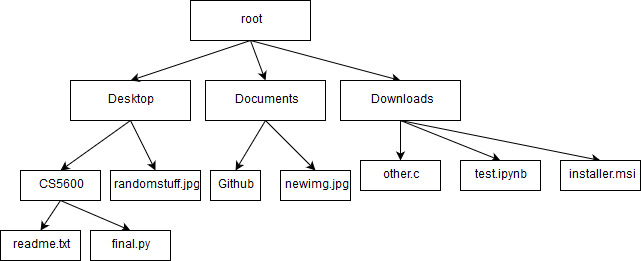
**II. Motivation**

The exponentially growth of data requires us to rethink the traditional file system. From experiences of managing a large number of files for analytic workloads, we observed that when the number of files in one directory exceeds some thousands, the performance of traditional operating system degrades dramatically. The intuition is that existing file systems did not take into consideration the exponentially growth of data due to advances of technologies such as cloud computing, Internet of Things, machine generated data, etc; and most importantly, the need to analyse all the data efficiently. This pushes us to implement a cache conscious metadata scheme for big data filesystems which is achieved by using the bloom filter.

**III. Filesystem tree**

We use a file-system tree to represent directories, subdirectories, and files. The data structure we use is an *n*-ary tree, so-called because a node of the tree could have any number of children.

For example, a snapshot of a directory might be represented in a *n*-ary tree like this:



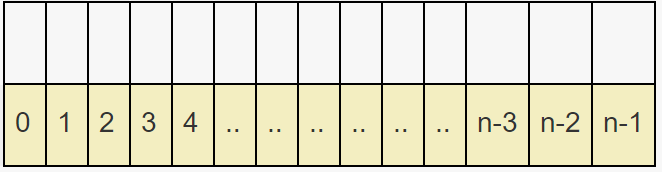
We use the simplest data structure to represent the disk since the improvements achieved by other data structures are not apparent while using the bloom filter. Querying the existence/location of data within a filesystem tree naively would require O(2n) time (for a tree of depth n), which slows down considerably for larger filesystems. Simple optimizations on the structure of tree used (eg. : radix tree, trie, splay tree, suffix tree) result in simple improvements in access time. However, these structures do not perform quite as efficiently when many disk accesses are made.

**IV. Why bloom filter?**

The bloom filter is a probabilistic data structure that is used to maintain/test the membership of an element in a set. It tells us if an element is definitely not in the set or maybe in the set.

The underlying data structure is a bit vector which is used to maintain the candidacy of an element.

Consider the following n bit vector:



To add an element to the filter, the element is simply hashed a few times and the index given by these hash functions are set to 1 in the bit vector.

Consider the following input: “The quick brown fox jumps over the lazy dog.”

murmur(“The quick brown fox jumps over the lazy dog.”) = 8

Which would result in the bit at index 8 being set to 1.

Consequently, it is sufficient to hash an element and check if the corresponding bits are set in the filter to check the membership of an element.

**V. Choosing a hash function, CRC32**

The CRC32 (32 Bit Cyclic Redundancy Check) is an error detection code used to check the integrity of the data (a checksum) which is commonly used for error detection in networks and storage devices.

The main reasoning behind choosing a CRC over a cryptographically strong encryption method is the constraint on length and constant time costs. Since speed is of the essence in querying operations, the hash function used must be computationally inexpensive.

To elaborate on this, consider the SHA-256 vs CRC32. Although SHA-256 is stronger (lesser collisions and harder to reverse), the bloom filter does not benefit from these advantages.

A cryptographic hash truncated to a length of 32 might easily collide for inputs that differ by one or two bytes. The CRC is designed towards reliably detecting such errors and is well protected against such cases.

**VI. Proposed Mechanism**

For every directory with a large number of files (>10,000), we maintain a bloom filter as the metadata for this path. When a file is requested, the file system traverses the tree from the root to the leaf to obtain this metadata, and determine the status of this file (exists or not). If there is a match, the filesystem continues to access that file; otherwise the filesystem goes to other locations or abort this request.

**VII. Fine tuning parameters**

Consider a bloom filter with k different hash functions, m bits in the filter and n elements already inserted. An obvious consequence of hashing the string more times is the accuracy of detecting positives. The trade-off being the speed of the bloom filter.

For an error rate e, the ideal number of hashes and filter bits are given by,

m = -n\*ln(e) / (ln(2)^2) the number of bits  
k = m/n \* ln(2) the number of hash functions

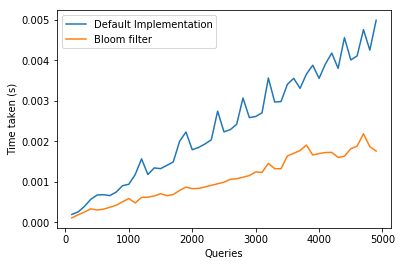
Although choosing these values appears ideal on first sight, m does not scale reliably across number of elements (number of files in the fstree). To avoid overfitting these values for a single n, the parameters were tested across multiple values of n and the best performing values were chosen.

The efficiency of the bloom filter can be measured by the lookup times for these (n, k, m) triples. For best tuning these values a common implementation is,

1. Choose the number of inputs (this must remain constant across multiple values to compare performances).
2. A value for the length of the filter is chosen.
3. An optimal value for the number of hashes is calculated.
4. If this falls within acceptable error rates, the process is complete. If it is not, the process is repeated with different filter lengths until an acceptable performance is obtained.

**VIII. Performance**

The experiment is run on MacBook Pro 2016 with 2.9 GHz Intel Core i5 processor and 8GB 2133 MHz LPDDR3. We generated 10,000 small files in a test directory whose names were collected from an English dictionary. We compared two ways to access the same random file: (1) using system call “access” to request it; (2) checking the metadata first and then request it depending on the feedback. Here is the graph:



The experiment shows that the effectiveness of file manipulation increases apparently even for not so many (10,000) files. And besides from query time being shorter, it is also more stable without sudden spikes. We argue this is a useful attribute for big data filesystems.