Bloom Filters in HBase

Things you can toggle

- HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL) to enable blooms per Column Family. Default = NONE
- Global
  - "io.hfile.bloom.enabled" = Kill switch in case something goes wrong. Default = True
  - "io.hfile.bloom.error.rate" = average false positive rate. default = 1% decrease rate by ½ (e.g. to .5%) == +1 bit per bloom entry
  - "io.hfile.bloom.max.fold" = guaranteed minimum fold rate. Most people should leave this alone. Default = 7, or can collapse to at least $^1/_{128}$ of original size. See commentary below.

HFile Storage

- Meta (retrieved on demand)
  - BLOOM_FILTER_META = Bloom Size, Hash Function, etc. Small size. Cached on StoreFile.Reader load
  - BLOOM_FILTER_DATA = Bloom data. Obtained on-demand, but stored in LRU cache, if present
- FileInfo (cached when HFile is loaded)
  - BLOOM_FILTER_TYPE = Is Bloom Row or Row+Col

Development Process

This document provides a quick survey of the status of bloom filters in HBase. It covers the unique properties of our HFile format that BloomFilters can exploit and the main optimization areas for Blooms.

Desired BloomFilter behavior:

1. Little wasted space. Maintaining bloom for 10Million keys only to insert 100 is wasteful.
2. Fast access. Read & Write path are almost the same
   a. Less Hash Functions
   b. Fast Hash Functions
   c. Smarter Hash Functions

Understanding HBase BloomFilter characteristics:

1. Known total KV count (N), but actual count can often be much lower
2. HFile.insert() [and hence, BloomFilter.add] commands are done in lexographically-increasing order

Thought process:

1. Initial path: Dynamic Blooms

a. Basic Idea: Instead of having 1 bloom with N rows, have M blooms with M/N rows
b. Why did we stop going down that path? In summary, because determining a miss requires checking every bloom.
    i. Misleading error rate. If every bloom has an E error rate, then actual error rate = M*E. Therefore, we would roughly need error rate of E/M per bloom or scale logarithmically.
        1. Either solution requires increased memory. E/2 à +1 bit per bloom entry.
    ii. Speed: bloom filter gets magnified by M on cache miss
c. Check out "Scalable Bloom Filters" if you are interested in this path; however Dynamic Blooms are mainly useful if don't know your upper bound, which we do.

2. Back to Static Blooms
    a. Decrease wasted space
        i. Since bloom bit to set = hash() % X, can cut bloom in ½ with OR operation if X%2 == 0
            1. Round up X so X % 2^F == 0, reduce bloom size up to $1/2^F$
            2. Bloom guaranteed > 50% full with correct value of F
            3. Super simple, easy to implement, & fast
            4. "io.hfile.bloom.max.fold" to toggle
        ii. *Future Idea: Bloom Block Index*
            1. *Keep Bloom keys in memory up to set size (or ratio like 1/100 of maxKeys)*
            2. *When size reached, create bloom with those keys & add entry to block index with start/end key*
            3. *When finalize called, add create last bloom*
            4. *Perfect sizing. No wasted space. Still need to binary search through block index on get, but can keep that cached*
        iii. Moved Bloom Data to LRU cache
    iv. Less Hash Functions
        i. Need minimum amount to maintain error rate.
        ii. No optimization here. For degradation, see Dynamic Blooms
    v. Fast Hash Functions
        i. Switched to Combinatoral Generation so you only do expensive hash creation 2x, no matter function count.
    vi. Smart Hash Functions
        i. Right now, every bloom bit set is random and mostly L1 cache misses for large blooms.
        ii. *Future Ideas: Cache-Efficient Bloom Filter*
            1. Locality-optimized hash functions
            2. Need to determine error bounds