LevelDB Study
Bloom Filter Analysis

Made by Kim Han Su

E-Mail: khs20010327@naver.com

UFTRACE

```
        [  1418] |   leveldb::Benchmark::Benchmark() {
        [  1418] |     leveldb::NewBloomFilterPolicy() {
0.412 us [  1418] |       operator new();
        [  1418] |       leveldb::_GLOBAL__N_1::BloomFilterPolicy::BloomFilterPolicy() {
0.052 us [  1418] |         leveldb::FilterPolicy::FilterPolicy();
0.288 us [  1418] |       } /* leveldb::_GLOBAL__N_1::BloomFilterPolicy::BloomFilterPolicy */
1.455 us [  1418] |     } /* leveldb::NewBloomFilterPolicy */
0.043 us [  1418] |     leveldb::WriteOptions::WriteOptions();
        [  1418] |     leveldb::BytewiseComparator() {
0.043 us [  1418] |       leveldb::NoDestructor::get();
```

NewBloomFilterPolicy

```
const FilterPolicy* NewBloomFilterPolicy(int bits_per_key) {
  return new BloomFilterPolicy(bits_per_key);
}
```

```
        [  1418] |   leveldb::Benchmark::Benchmark() {
        [  1418] |     leveldb::NewBloomFilterPolicy() {
0.412 us [  1418] |       operator new();
        [  1418] |       leveldb::_GLOBAL__N_1::BloomFilterPolicy::BloomFilterPolicy() {
```

FilterPolicy

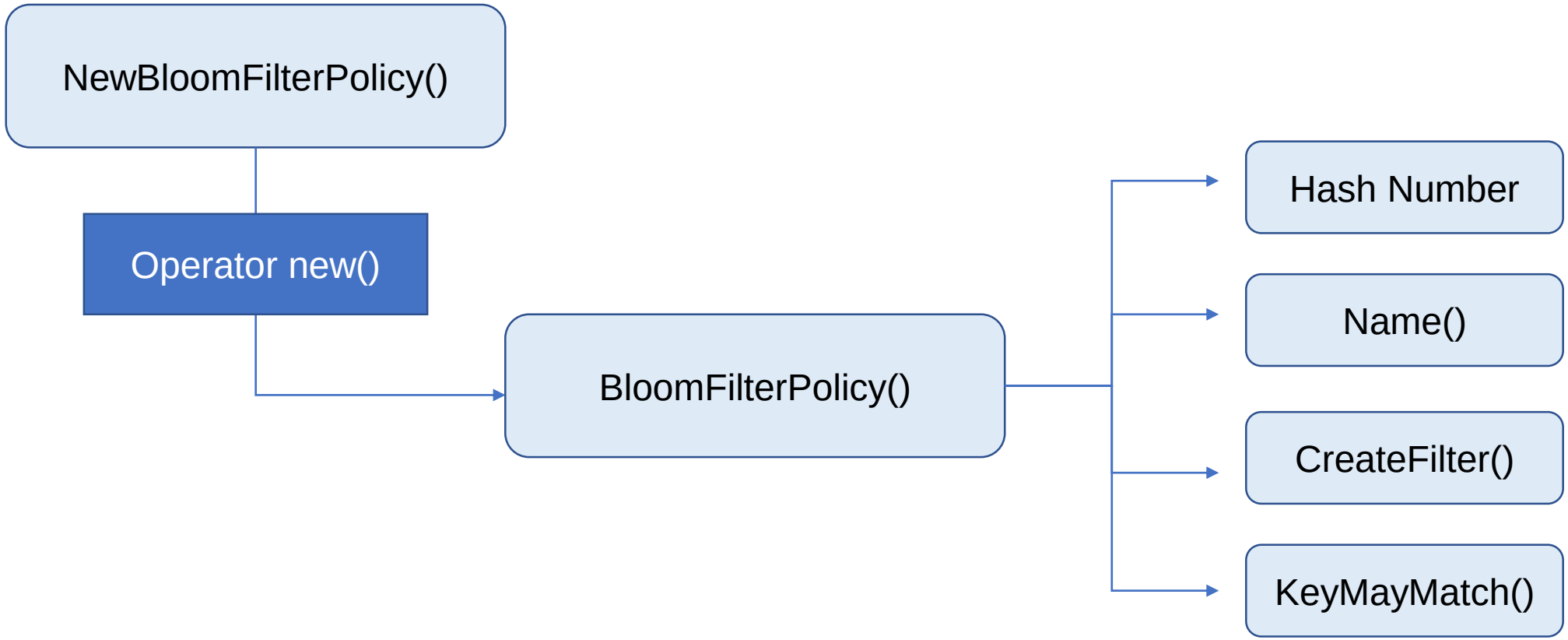
```
class LEVELDB_EXPORT FilterPolicy {
 public:
  virtual ~FilterPolicy();

  // Return the name of this policy.  Note that if the filter encoding
  // changes in an incompatible way, the name returned by this method
  // must be changed.  Otherwise, old incompatible filters may be
  // passed to methods of this type.
  virtual const char* Name() const = 0;

  virtual void CreateFilter(const Slice* keys, int n,
                            std::string* dst) const = 0;

  virtual bool KeyMayMatch(const Slice& key, const Slice& filter) const = 0;
};
```

CodeFlow


NewBloomFilterPolicy()
Operator new()
BloomFilterPolicy()
Hash Number
Name()
CreateFilter()
KeyMayMatch()

Hash Number & Name()

```
class BloomFilterPolicy : public FilterPolicy {
 public:
  explicit BloomFilterPolicy(int bits_per_key) : bits_per_key_(bits_per_key) {
    // We intentionally round down to reduce probing cost a little bit
    k_ = static_cast<size_t>(bits_per_key * 0.69);  // 0.69 =~ ln(2)
    if (k_ < 1) k_ = 1;
    if (k_ > 30) k_ = 30;
  }

  const char* Name() const override { return "leveldb.BuiltinBloomFilter2"; }
```

CreateFilter

```
void CreateFilter(const Slice* keys, int n, std::string* dst) const override {
  // Compute bloom filter size (in both bits and bytes)
  size_t bits = n * bits_per_key_;

  // For small n, we can see a very high false positive rate.  Fix it
  // by enforcing a minimum bloom filter length.
  if (bits < 64) bits = 64;

  size_t bytes = (bits + 7) / 8;
  bits = bytes * 8;
```

CreateFilter

```
  const size_t init_size = dst->size();
  dst->resize(init_size + bytes, 0);
  dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter
  char* array = &(*dst)[init_size];
  for (int i = 0; i < n; i++) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    uint32_t h = BloomHash(keys[i]);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k_; j++) {
      const uint32_t bitpos = h % bits;
      array[bitpos / 8] |= (1 << (bitpos % 8));
      h += delta;
    }
  }
}
```

CreateFilter

```
  const size_t init_size = dst->size();
  dst->resize(init_size + bytes, 0);
  dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter
  char* array = &(*dst)[init_size];
  for (int i = 0; i < n; i++) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    uint32_t h = BloomHash(keys[i]);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k_; j++) {
      const uint32_t bitpos = h % bits;
      array[bitpos / 8] |= (1 << (bitpos % 8));
      h += delta;
    }
  }
}
```

BloomHash & Hash

```
namespace {
static uint32_t BloomHash(const Slice& key) {
  return Hash(key.data(), key.size(), 0xbc9f1d34);
}
```

```
uint32_t Hash(const char* data, size_t n, uint32_t seed) {
  // Similar to murmur hash
  const uint32_t m = 0xc6a4a793;
  const uint32_t r = 24;
  const char* limit = data + n;
  uint32_t h = seed ^ (n * m);
```

CreateFilter

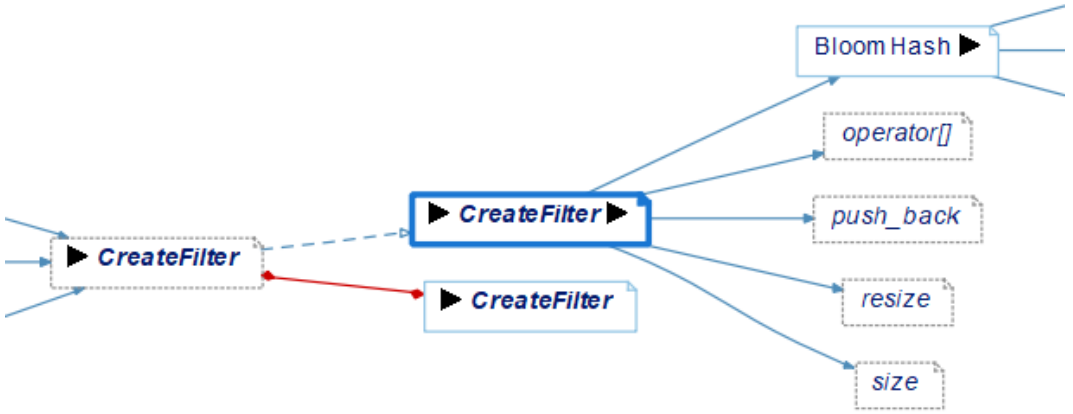
```
  const size_t init_size = dst->size();
  dst->resize(init_size + bytes, 0);
  dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter
  char* array = &(*dst)[init_size];
  for (int i = 0; i < n; i++) {
    // Use double-hashing to generate a sequence of hash values.
    // See analysis in [Kirsch,Mitzenmacher 2006].
    uint32_t h = BloomHash(keys[i]);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k_; j++) {
      const uint32_t bitpos = h % bits;
      array[bitpos / 8] |= (1 << (bitpos % 8));
      h += delta;
    }
  }
}
```

KeyMayMatch

```
  uint32_t h = BloomHash(key);
  const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
  for (size_t j = 0; j < k; j++) {
    const uint32_t bitpos = h % bits;
    if ((array[bitpos / 8] & (1 << (bitpos % 8))) == 0) return false;
    h += delta;
  }
  return true;
}
```

InternalFilter?

```
          [  1471] |      leveldb::DBImpl::DBImpl() {
0.039 us  [  1471] |        leveldb::DB::DB();
          [  1471] |        leveldb::InternalKeyComparator::InternalKeyComparator() {
0.041 us  [  1471] |          leveldb::Comparator::Comparator();
0.152 us  [  1471] |        } /* leveldb::InternalKeyComparator::InternalKeyComparator */
          [  1471] |        leveldb::InternalFilterPolicy::InternalFilterPolicy() {
0.034 us  [  1471] |          leveldb::FilterPolicy::FilterPolicy();
0.143 us  [  1471] |        } /* leveldb::InternalFilterPolicy::InternalFilterPolicy */
          [  1471] |        leveldb::SanitizeOptions() {
0.052 us  [  1471] |          leveldb::ClipToRange();
```



InternalFilterPolicy

```
// Filter policy wrapper that converts from internal keys to user keys
class InternalFilterPolicy : public FilterPolicy {
 private:
  const FilterPolicy* const user_policy_;

 public:
  explicit InternalFilterPolicy(const FilterPolicy* p) : user_policy_(p) {}
  const char* Name() const override;
  void CreateFilter(const Slice* keys, int n, std::string* dst) const override;
  bool KeyMayMatch(const Slice& key, const Slice& filter) const override;
};
```

InternalFilterPolicy

```
void InternalFilterPolicy::CreateFilter(const Slice* keys, int n,
                                        std::string* dst) const {
  // We rely on the fact that the code in table.cc does not mind us
  // adjusting keys[].
  Slice* mkey = const_cast<Slice*>(keys);
  for (int i = 0; i < n; i++) {
    mkey[i] = ExtractUserKey(keys[i]);
    // TODO(sanjay): Suppress dups?
  }
  user_policy_->CreateFilter(keys, n, dst);
}
```

ExtractUserKey

```
// Returns the user key portion of an internal key.
inline Slice ExtractUserKey(const Slice& internal_key) {
  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}
```

ExtractUserKey

```
// Returns the user key portion of an internal key.
inline Slice ExtractUserKey(const Slice& internal_key) {
  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}
```

ExtractUserKey

```
// Returns the user key portion of an internal key.
inline Slice ExtractUserKey(const Slice& internal_key) {
  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}
```

- Key
  - userkey: passed from user, in Slice format
  - InternalParsedKey: userkey + seqNum + valuetype

Google's original internal key

Google's original leveldb internal keys comprise of four components:

- total internal key size
- user's binary key
- 7 byte sequence number
- 1 byte type code

The internal key size includes the user's binary key, sequence number, and type code.

InternalFilterPolicy

```
const char* InternalFilterPolicy::Name() const { return user_policy_->Name(); }

void InternalFilterPolicy::CreateFilter(const Slice* keys, int n,
                                        std::string* dst) const {
  // We rely on the fact that the code in table.cc does not mind us
  // adjusting keys[].
  Slice* mkey = const_cast<Slice*>(keys);
  for (int i = 0; i < n; i++) {
    mkey[i] = ExtractUserKey(keys[i]);
    // TODO(sanjay): Suppress dups?
  }
  user_policy_->CreateFilter(keys, n, dst);
}

bool InternalFilterPolicy::KeyMayMatch(const Slice& key, const Slice& f) const {
  return user_policy_->KeyMayMatch(ExtractUserKey(key), f);
}
```

QnA