LevelDB Study
Bloom Filter Analysis

Made by Kim Han Su

E-Mail: khs20010327@naver.com

DANKOOK UNIVERSITY

Dankook University
System Software Laboratory

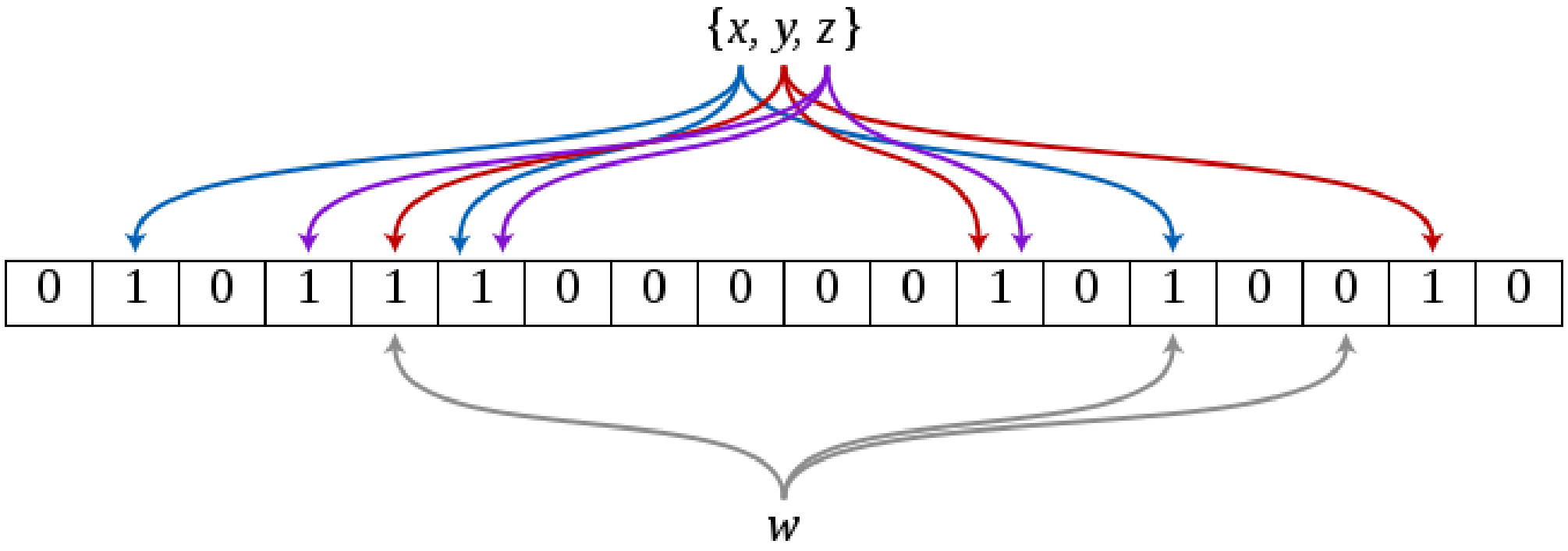
Contents

- About double-hashing
- Code flow of bloom filter write

Double hashing

```
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;
  }
```

What is double hashing?



Bloom filter needs k hash result per 1 key

[Kirsch, Mitzenmacher 2006]

Less Hashing, Same Performance: Building a Better Bloom Filter

Abstract. A standard technique from the hashing literature is to use two hash functions $h_1(x)$ and $h_2(x)$ to simulate additional hash functions of the form $\boxed{g_i(x) = h_1(x) + ih_2(x).}$ We demonstrate that this technique can be usefully applied to Bloom filters and related data structures. Specifically, only two hash functions are necessary to effectively implement a Bloom filter without any loss in the asymptotic false positive probability. This leads to less computation and potentially less need for randomness in practice.

2 hash function for LevelDB

$$g_i(x) = h_1(x) + ih_2(x).$$

```
uint32_t h = BloomHash(keys[i]);
```

```
const uint32_t delta = (h >> 17) | (h << 15);
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);
```

Hash 2 example

```
h=3828766811
```

h = 11100100001101100101100001011011 (32bits)

```
uint32_t h = BloomHash(keys[i]);
```

Hash 2 example

```
h=3828766811
delta=741208603
```

```
(h >> 17) | (h << 15)
```

h = 11100100001101100101100001011011 (32bits)

h>>17 = 00000000000000001110010000110 11

Hash 2 example

```
h=3828766811
delta=741208603
```

```
(h >> 17) | (h << 15)
```

h = 11100100001101100101100001011011 (32bits)

h<<15 = 00101100001011011000000000000000

Hash 2 example

```
h=3828766811
delta=741208603
```

```
(h >> 17) | (h << 15)
```

h = 11100100001101100101101000010110 11 (32bits)

h>>17 = 00000000000000001110010000110 11

h<<15 = 00101100001011011000000000000000

h>>17 | h<<15 = 00101100001011011110010000110 11

CreateFilter (Write)

```
uint32_t h = BloomHash(keys[i]);
const uint32_t delta = (h >> 17) | (h << 15);
for (size_t j = 0; j < k_; j++) {
  const uint32_t bitpos = h % bits;
  array[bitpos / 8] |= (1 << (bitpos % 8));
  h += delta;
```

00000000 ➡ 00010000

KeyMayMatch (Read)

```
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;
```

Main Function

Instant Search for 'main': results found in 97 file(s). 3:42:40 오후

- benchmarks\db_bench.cc
- benchmarks\db_bench_sqlite3.cc
- benchmarks\db_bench_tree_db.cc
- build\CMakeFiles\3.16.3\CompilerIdCXX\CMakeCXXCompilerId.cpp
- build\CMakeFiles\3.16.3\CompilerIdC\CMakeCCompilerId.c
- db\c_test.c
- db\leveldbutil.cc
- doc\benchmark.html
- third_party\benchmark\bindings\python\google_benchmark__init__.py
- third_party\benchmark\cmake\gnu_posix_regex.cpp
- third_party\benchmark\cmake\posix_regex.cpp

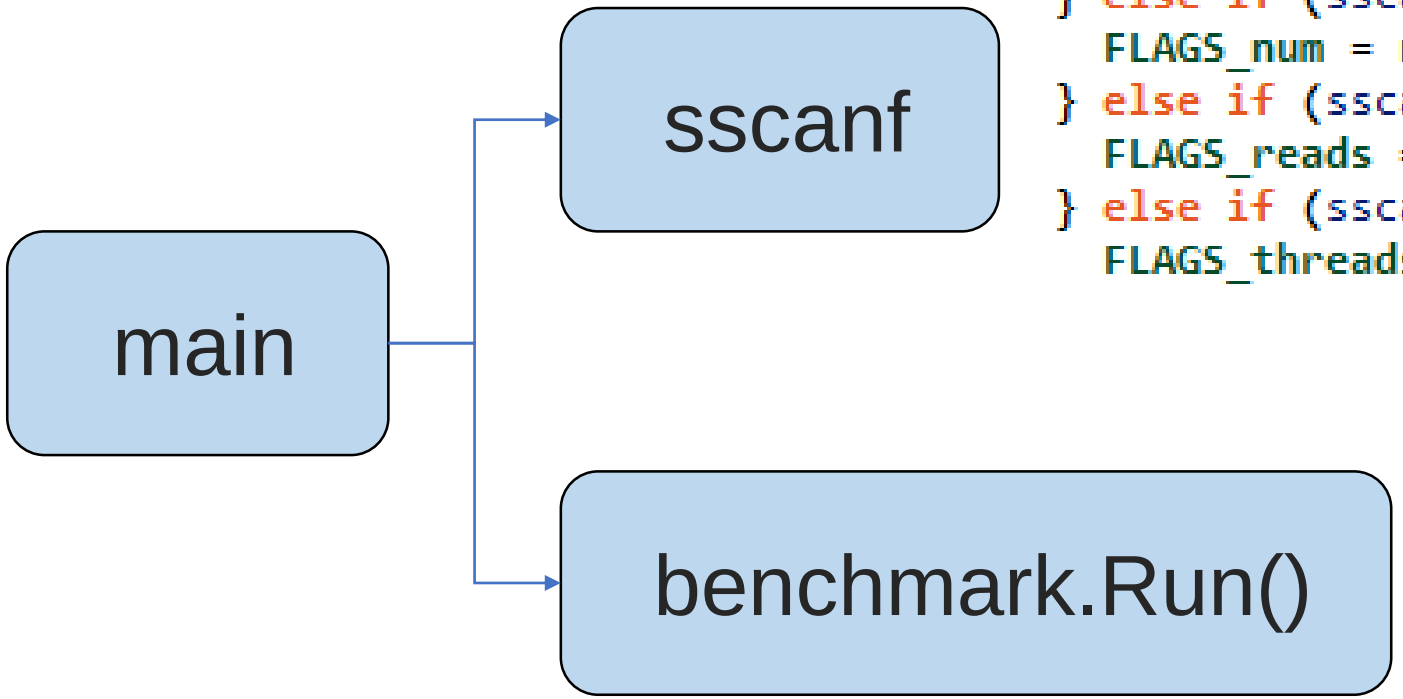
Main Function

Instant Search for 'main': results found in 97 file(s). 3:42:40 오후

- benchmarks\db_bench.cc
- benchmarks\db_bench_sqlite3.cc
- benchmarks\db_bench_tree_db.cc
- build\CMakeFiles\3.16.3\CompilerIdCXX\CMakeCXXCompilerId.c
- build\CMakeFiles\3.16.3\CompilerIdC\CMakeCCompilerId.c
- db\c_test.c
- db\leveldbutil.cc
- doc\benchmark.html
- third_party\benchmark\bindings\python\google_benchmark__init__.py
- third_party\benchmark\cmake\gnu_posix_regex.cpp
- third_party\benchmark\cmake\posix_regex.cpp

```
# target to build an object file
benchmarks/db_bench.cc.o:
	$(MAKE) -f CMakeFiles/db_bench.d
.PHONY : benchmarks/db_bench.cc.o
```

db_bench.cc




```
} else if (sscanf(argv[i], "--num=%d%c", &n, &junk) :
  FLAGS_num = n;
} else if (sscanf(argv[i], "--reads=%d%c", &n, &junk
  FLAGS_reads = n;
} else if (sscanf(argv[i], "--threads=%d%c", &n, &ju
  FLAGS_threads = n;
```

```
leveldb::Benchmark benchmark;
benchmark.Run();
return 0;
```

Benchmark.Run()

benchmark.Run()

```
leveldb::Benchmark benchmark;
benchmark.Run();
return 0;
```

class Benchmark

Benchmark()

Run()

Class Benchmark

class Benchmark

```
class Benchmark {
 private:
  Cache* cache_;
  const FilterPolicy* filter_policy_;
  DB* db_;
  int num_;
  int value_size_;
  int entries_per_batch_;
  WriteOptions write_options_;
  int reads_;
  int heap_counter_;
  CountComparator count_comparator_;
  int total_thread_count_;
```

Constructor of Benchmark

Benchmark()

```
public:
 Benchmark()
     : cache_(FLAGS_cache_size >= 0 ? NewLRUCache(FLAGS_cache_size) : nullptr),
       filter_policy_(FLAGS_bloom_bits >= 0
                      ? NewBloomFilterPolicy(FLAGS_bloom_bits)
                      : nullptr),
```

NewBloomFilterPolicy

NewBloomFilterPolicy()

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

```
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;
  }
```

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;
};
```

Benchmark.Run()

benchmark.Run()

```
leveldb::Benchmark benchmark;
benchmark.Run();
return 0;
```

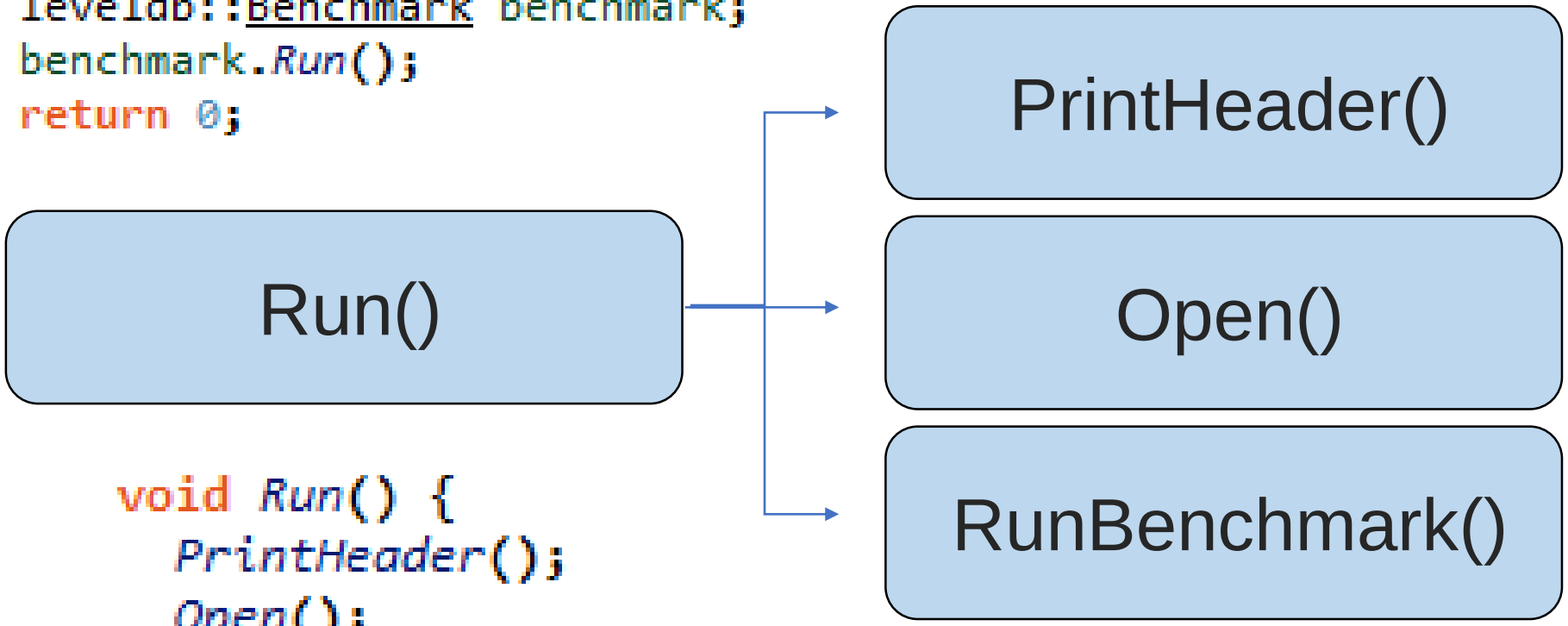
class Benchmark

Benchmark()

Run()

Run()

```
leveldb::Benchmark benchmark;
benchmark.Run();
return 0;
```




```
void Run() {
  PrintHeader();
  Open();
```

PrintHeader()

PrintHeader()

```
solid@hansu272-68b87755d6-dsj4s:~/debug/leveldb/build$ uftrace record ./db_bench --benchmarks="fil
ts" --bloom_bits=10 --num=700
LevelDB:    version 1.23
Date:       Sat Aug  6 03:08:17 2022
CPU:        40 * Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
CPUCache:   14080 KB
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compression)
Entries:    700
RawSize:    0.1 MB (estimated)
FileSize:   0.0 MB (estimated)
```

Open()

```
void Open() {
  assert(db_ == nullptr);
  Options options;
  options.env = g_env;
  options.create_if_missing = !FLAGS_use_existing_db;
  options.block_cache = cache_;
  options.write_buffer_size = FLAGS_write_buffer_size;
  options.max_file_size = FLAGS_max_file_size;
  options.block_size = FLAGS_block_size;
  if (FLAGS_comparisons) {
    options.comparator = &count_comparator_;
  }
  options.max_open_files = FLAGS_open_files;
  options.filter_policy = filter_policy_;
  options.reuse_logs = FLAGS_reuse_logs;
  Status s = DB::Open(options, FLAGS_db, &db_);
  if (!s.ok()) {
    std::fprintf(stderr, "open error: %s\n", s.ToString().c_str());
    std::exit(1);
  }
}
```

DB::Open()

```
Status DB::Open(const Options& options, const std::string& dbname, DB** dbptr) {
  *dbptr = nullptr;

  DBImpl* impl = new DBImpl(options, dbname);
  impl->mutex_.Lock();
  VersionEdit edit;


  if (s.ok()) {
    impl->RemoveObsoleteFiles();
    impl->MaybeScheduleCompaction();
  }
```

MaybeSchduleCompaction()

Maybe Schedule Compaction()

NeedsCompaction()

BGWork()

```
void DBImpl::MaybeScheduleCompaction() {
  mutex_.AssertHeld();
  if (background_compaction_scheduled_) {
    // Already scheduled
  } else if (shutting_down_.load(std::memory_order_a
    // DB is being deleted; no more background compa
  } else if (!bg_error_.ok()) {
    // Already got an error; no more changes
  } else if (imm_ == nullptr && manual_compaction_ =
             !versions_->NeedsCompaction()) {
    // No work to be done
  } else {
    background_compaction_scheduled_ = true;
    env_->Schedule(&DBImpl::BGWork, this);
  }
}
```

CodeFlow

- MaybeSchedulCompaction() -> BGWork() -> BackgroundCall()

-> BackgroundCompaction -> CompactMemtable()

-> WriteLevel0Table -> BuildTable() -> Add() / Finish() /Flush()

-> Startblock() / Finish() -> GenerateFilter()

NeedsCompaction

```
leveldb::DBImpl::MaybeScheduleCompaction() {
    leveldb::port::Mutex::AssertHeld();
  std::atomic::load() {
      std::operator&();
    } /* std::atomic::load */
    leveldb::Status::ok();
    leveldb::VersionSet::NeedsCompaction();
```

NeedsCompaction

```
} else if (imm_ == NULL &&
          manual_compaction_ == NULL &&
          !versions_->NeedsCompaction()) {
  // No work to be done
```

```
uftrace record ./db_bench --benchmarks="fillseq,compact"
```

```
0.377 us [  2734] |          } /* std::mutex::unlock */
0.457 us [  2734] |        } /* leveldb::port::Mutex::Unlock */
       [  2734] |      leveldb::DBImpl::BGWork() {
       [  2734] |        leveldb::DBImpl::BackgroundCall() {
       [  2734] |          leveldb::MutexLock::MutexLock() {
```

CodeFlow

- MaybeSchedulCompaction() -> BGWork() -> BackgroundCall()

-> BackgroundCompaction -> CompactMemtable()

-> WriteLevel0Table -> BuildTable() -> Add() / Finish() /Flush()

-> Startblock() / Finish() -> GenerateFilter()

GenerateFilter

```
void FilterBlockBuilder::GenerateFilter() {
  const size_t num_keys = start_.size();
  if (num_keys == 0) {
    // Fast path if there are no keys for this filter
    filter_offsets_.push_back(result_.size());
    return;
  }

  // Make list of keys from flattened key structure
  start_.push_back(keys_.size());  // Simplify length computation
  tmp_keys_.resize(num_keys);
  for (size_t i = 0; i < num_keys; i++) {
    const char* base = keys_.data() + start_[i];
    size_t length = start_[i + 1] - start_[i];
    tmp_keys_[i] = Slice(base, length);
  }

  // Generate filter for current set of keys and append to result_.
  filter_offsets_.push_back(result_.size());
  policy_->CreateFilter(&tmp_keys_[0], static_cast<int>(num_keys), &result_);

  tmp_keys_.clear();
  keys_.clear();
  start_.clear();
}
```

KeyMayMatch

```
bool KeyMayMatch(const Slice& key, const Slice& bloom_filter) const override {
  const size_t len = bloom_filter.size();
  if (len < 2) return false;

  const char* array = bloom_filter.data();
  const size_t bits = (len - 1) * 8;

  // Use the encoded k so that we can read filters generated by
  // bloom filters created using different parameters.
  const size_t k = array[len - 1];
  if (k > 30) {
    // Reserved for potentially new encodings for short bloom filters.
    // Consider it a match.
    return true;
  }
```

Question