LevelDB Study
Bloom Filter Analysis

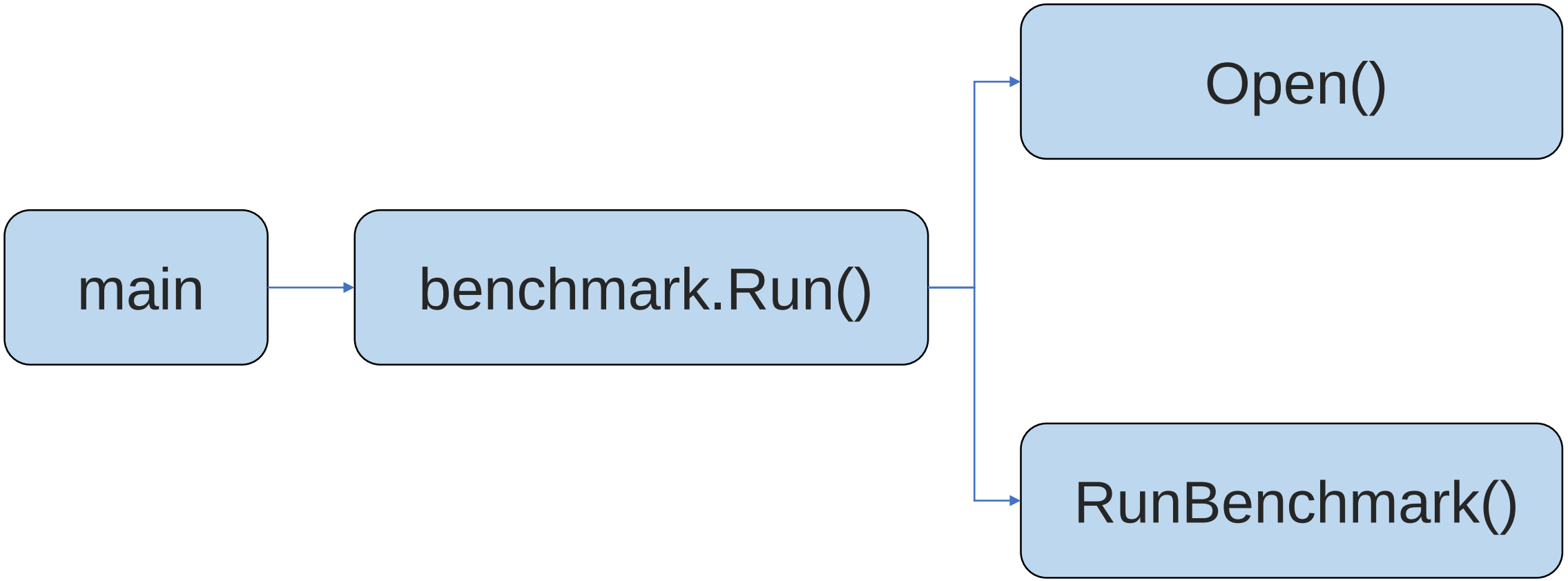
Made by Kim Han Su

E-Mail: khs20010327@naver.com

Contents

- Code flow of bloom filter read
- More about db_bench

db_bench.cc



RunBenchmark()

```
if (method != nullptr) {
  RunBenchmark(num_threads, name, method);
}
```

Num_threads

```
        if (method != nullptr) {
          RunBenchmark(num_threads, name, method);
        }
```

```
int num_threads = FLAGS_threads;
```

```
} else if (name == Slice("readwhilewriting")) {
    num_threads++;  // Add extra thread for writing
    method = &Benchmark::ReadWhileWriting;
```

Name

```
if (method != nullptr) {
  RunBenchmark(num_threads, name, method);
}
```

```
const char* benchmarks = FLAGS_benchmarks;
while (benchmarks != nullptr) {
  const char* sep = strchr(benchmarks, ',');
  Slice name;
  if (sep == nullptr) {
    name = benchmarks;
    benchmarks = nullptr;
  } else {
    name = Slice(benchmarks, sep - benchmarks);
    benchmarks = sep + 1;
  }
```

Method

```
if (method != nullptr) {
  RunBenchmark(num_threads, name, method);
}
```

```
void (Benchmark::*method)(ThreadState*) = nullptr;
```

```
} else if (name == Slice("readseq")) {
  method = &Benchmark::ReadSequential;
} else if (name == Slice("readreverse")) {
  method = &Benchmark::ReadReverse;
} else if (name == Slice("readrandom")) {
  method = &Benchmark::ReadRandom;
```

RunBenchmark()

```
void RunBenchmark(int n, Slice name,
                  void (Benchmark::*method)(ThreadState*)) {
  SharedState shared(n);

  ThreadArg* arg = new ThreadArg[n];
  for (int i = 0; i < n; i++) {
    arg[i].bm = this;
    arg[i].method = method;
    arg[i].shared = &shared;
    ++total_thread_count_;
    arg[i].thread = new ThreadState(i, /*seed=*/1000 + total_thread_count_);
    arg[i].thread->shared = &shared;
    g_env->StartThread(ThreadBody, &arg[i]);
  }
```

```
virtual void StartThread(void (*function)(void* arg), void* arg) = 0;
```

ThreadBody()

```
static void ThreadBody(void* v) {
  ThreadArg* arg = reinterpret_cast<ThreadArg*>(v);
  SharedState* shared = arg->shared;
  ThreadState* thread = arg->thread;
  {
    MutexLock l(&shared->mu);
    shared->num_initialized++;
    if (shared->num_initialized >= shared->total) {
      shared->cv.SignalAll();
    }
    while (!shared->start) {
      shared->cv.Wait();
    }
  }
  thread->stats.Start();
  (arg->bm->*(arg->method))(thread);
  thread->stats.Stop();
```

Code Flow Chart

Function	purpose
Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

ReadRandom

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

```
void ReadRandom(ThreadState* thread) {
  ReadOptions options;
  std::string value;
  int found = 0;
  KeyBuffer key;
  for (int i = 0; i < reads_; i++) {
    const int k = thread->rand.Uniform(FLAGS_num);
    key.Set(k);
    if (db_->Get(options, key.slice(), &value).ok()) {
      found++;
    }
    thread->stats.FinishedSingleOp();
  }
  char msg[100];
  std::snprintf(msg, sizeof(msg), "(%d of %d found)", found, num_);
  thread->stats.AddMessage(msg);
}
```

```
Status s = DB::Open(options, FLAGS_db, &db_);
```

DBImpl::Get()

```
Status DBImpl::Get(const ReadOptions& options, const Slice& key,
                   std::string* value) {
  if (mem->Get(lkey, value, &s)) {
    // Done
  } else if (imm != nullptr && imm->Get(lkey, value, &s)) {
    // Done
  } else {
    s = current->Get(options, lkey, value, &stats);
    have_stat_update = true;
  }
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

Version::Get()

```
Status Version::Get(const ReadOptions& options, const LookupKey& k,
                    std::string* value, GetStats* stats) {
  stats->seek_file = nullptr;
  stats->seek_file_level = -1;
  struct State { ...
  ForEachOverlapping(state.saver.user_key, state.ikey, &state, &State::Match);
  return state.found ? state.s : Status::NotFound(Slice());
}
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

ForEachOverlapping()

```
void Version::ForEachOverlapping(Slice user_key, Slice internal_key, void* arg,
                                 bool (*func)(void*, int, FileMetaData*)) {
  const Comparator* ucmp = vset_->icmp_.user_comparator();

  // Search level-0 in order from newest to oldest.
  std::vector<FileMetaData*> tmp;
  tmp.reserve(files_[0].size());
  for (uint32_t i = 0; i < files_[0].size(); i++) {
    FileMetaData* f = files_[0][i];
    if (ucmp->Compare(user_key, f->smallest.user_key()) >= 0 &&
        ucmp->Compare(user_key, f->largest.user_key()) <= 0) {
      tmp.push_back(f);
    }
  }
  if (!tmp.empty()) { ...

  // Search other levels.
  for (int level = 1; level < config::kNumLevels; level++) {
    size_t num_files = files_[level].size();
    if (num_files == 0) continue;
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

Get::State::Match()

```
static bool Match(void* arg, int level, FileMetaData* f) {
  State* state = reinterpret_cast<State*>(arg);

  state->s = state->vset->table_cache_->Get(*state->options, f->number,
                                            f->file_size, state->ikey,
                                            &state->saver, SaveValue);

  if (!state->s.ok()) {
    state->found = true;
    return false;
  }
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

TableCache::Get()

```
Status TableCache::Get(const ReadOptions& options, uint64_t file_number,
                       uint64_t file_size, const Slice& k, void* arg,
                       void (*handle_result)(void*, const Slice&,
                                             const Slice&)) {
  Cache::Handle* handle = nullptr;
  Status s = FindTable(file_number, file_size, &handle);
  if (s.ok()) {
    Table* t = reinterpret_cast<TableAndFile*>(cache_->Value(handle))->table;
    s = t->InternalGet(options, k, arg, handle_result);
    cache_->Release(handle);
  }
  return s;
}
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

Table::InternalGet()

```
Status Table::InternalGet(const ReadOptions& options, const Slice& k, void* arg,
                          void (*handle_result)(void*, const Slice&,
                                                const Slice&)) {
  Status s;
  Iterator* iiter = rep_->index_block->NewIterator(rep_->options.comparator);
  iiter->Seek(k);
  if (iiter->Valid()) {
    Slice handle_value = iiter->value();
    FilterBlockReader* filter = rep_->filter;
    BlockHandle handle;
    if (filter != nullptr && handle.DecodeFrom(&handle_value).ok() &&
        !filter->KeyMayMatch(handle.offset(), k)) {
      // Not found
    } else {
      Iterator* block_iter = BlockReader(this, options, iit
      block_iter->Seek(k);
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

FilterBlockReader::KeyMayMatch()

```
bool FilterBlockReader::KeyMayMatch(uint64_t block_offset, const Slice& key) {
  uint64_t index = block_offset >> base_lg_;
  if (index < num_) {
    uint32_t start = DecodeFixed32(offset_ + index * 4);
    uint32_t limit = DecodeFixed32(offset_ + index * 4 + 4);
    if (start <= limit && limit <= static_cast<size_t>(offset_ - data_)) {
      Slice filter = Slice(data_ + start, limit - start);
      return policy_->KeyMayMatch(key, filter);
    } else if (start == limit) {
      // Empty filters do not match any keys
      return false;
    }
  }
  return true;  // Errors are treated as potential
}
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

BloomFilterPolicy::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;
  }

  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;
}
```

Benchmark::ReadRandom()	데이터베이스 연결
DBImpl::Get()	Memtable, Immemtable, SStable
Version::Get()	SStable 탐색
Version::ForEachOverlapping()	각 레벨 탐색
Version::Get::State::Match()	키 존재 여부 bool 리턴
TableCache::Get()	테이블 탐색
Table::InternalGet()	키 존재 확인시 데이터 블록 Read
FilterBlockReader::KeyMayMatch()	필터 블록 탐색
BloomFilterPolicy::KeyMayMatch()	블룸 필터 탐색

Shell Script

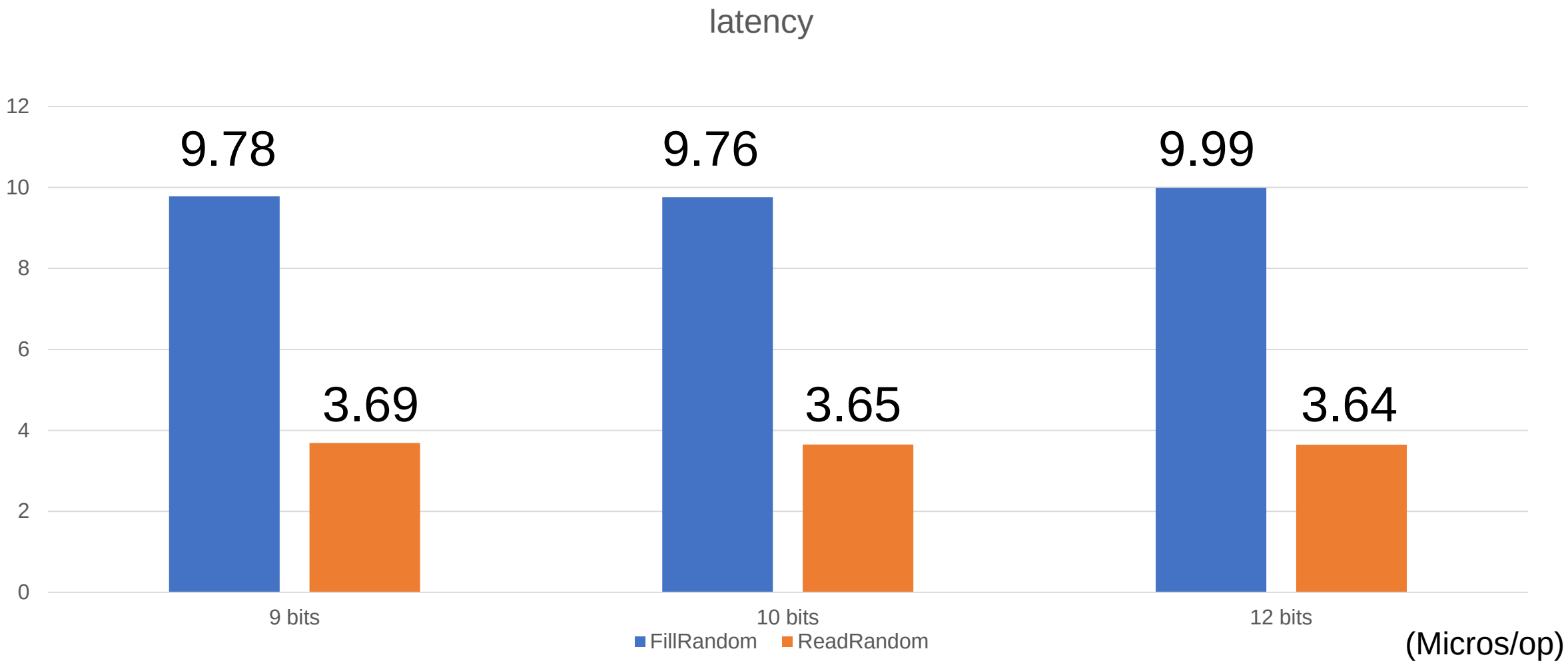
```
solid@hansu272-68b87755d6-dsj4s:~/release/leveldb/build$ bash a.sh
Count 1
./db_bench            --use_existing_db=0            --histogram=1            --compression_ratio=1            --benchm
arks=fillrandom,readrandom
            --num=1000000            --value_size=128            --bloom_bits=10


LevelDB:    version 1.23
Date:       Mon Aug 22 21:20:33 2022
CPU:        40 * Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
CPUCache:   14080 KB
Keys:       16 bytes each
Values:     128 bytes each (128 bytes after compression)
Entries:    1000000
RawSize:    137.3 MB (estimated)
FileSize:   137.3 MB (estimated)
WARNING: Snappy compression is not enabled
------------------------------------------------
fillrandom   :       4.421 micros/op;   31.1 MB/s
Microseconds per op:
Count: 1000000  Average: 4.4210  StdDev: 38.47
Min: 1.0000  Median: 3.5580  Max: 14784.0000
------------------------------------------------------
[       1,       2 )       31   0.003%   0.003%
[       2,       3 )   231643  23.164%  23.167% #####
[       3,       4 )   480894  48.089%  71.257% ##########
[       4,       5 )   184857  18.486%  89.743% ####
[       5,       6 )    57485   5.748%  95.491% #
[       6,       7 )    17539   1.754%  97.245%
[       7,       8 )     8547   0.855%  98.100%
[       8,       9 )     6734   0.673%  98.773%
```

Hypothesis

- LevelDB는 ln2 * bits_per_key개의 해시 함수를 사용
- ->Bits per key 값이 10이라면 이론상 6.9개의 해시 함수를 사용해야함
- ->실제론 6개를 사용
- 9 -> 9 X 0.69 = 6.21 -> 6
- 12 -> 12 X 0.69 = 8.28 -> 8

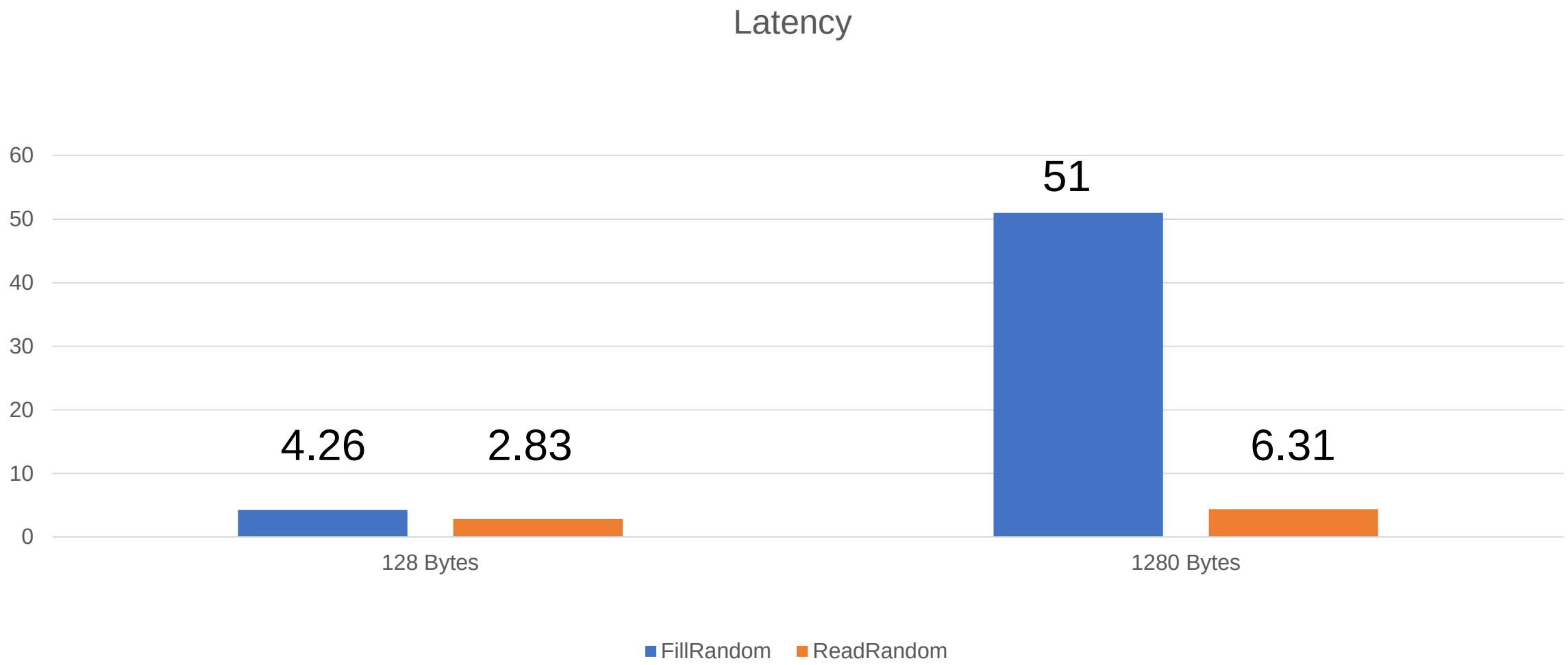
Result


latency
12
10
8
6
4
2
0
9.78
3.69
9.76
3.65
9.99
3.64
9 bits
10 bits
12 bits
FillRandom
ReadRandom
(Micros/op)

Hypothesis

- Bloom filter로 인한 쓰기 성능 감소는 key의 개수에 비례하고 읽기 성능 증가는 읽기 시간에 비례할 것이다.
- 그렇다면 Value_size만을 키웠을 때 Bloom Filter로 인한 쓰기 성능 감소는 동일하고 읽기 성능의 향상 폭은 커질 것이다

Result


Latency
60
50
40
30
20
10
0
4.26
2.83
51
6.31
128 Bytes
1280 Bytes
FillRandom
ReadRandom

Result

Value_size	128 bytes	128 bytes	1280 bytes	1280 bytes
Benchmarks	FillRandom	ReadRandom	FillRandom	ReadRandom
블룸 필터 미사용	4.26 (micros/op)	2.83	51	6.31
블룸 필터 사용	4.55	2.64	55	5.71

Result

Value_size	128 bytes	128 bytes	1280 bytes	1280 bytes
Benchmarks	FillRandom	ReadRandom	FillRandom	ReadRandom
블룸 필터 미사용	4.26 (micros/op)	2.83	51	6.31
블룸 필터 사용	4.55	2.64	55	5.71

-Value_size를 10배로 늘리자 쓰기 시간은 12배, 읽기 시간은 2.2배 증가했다

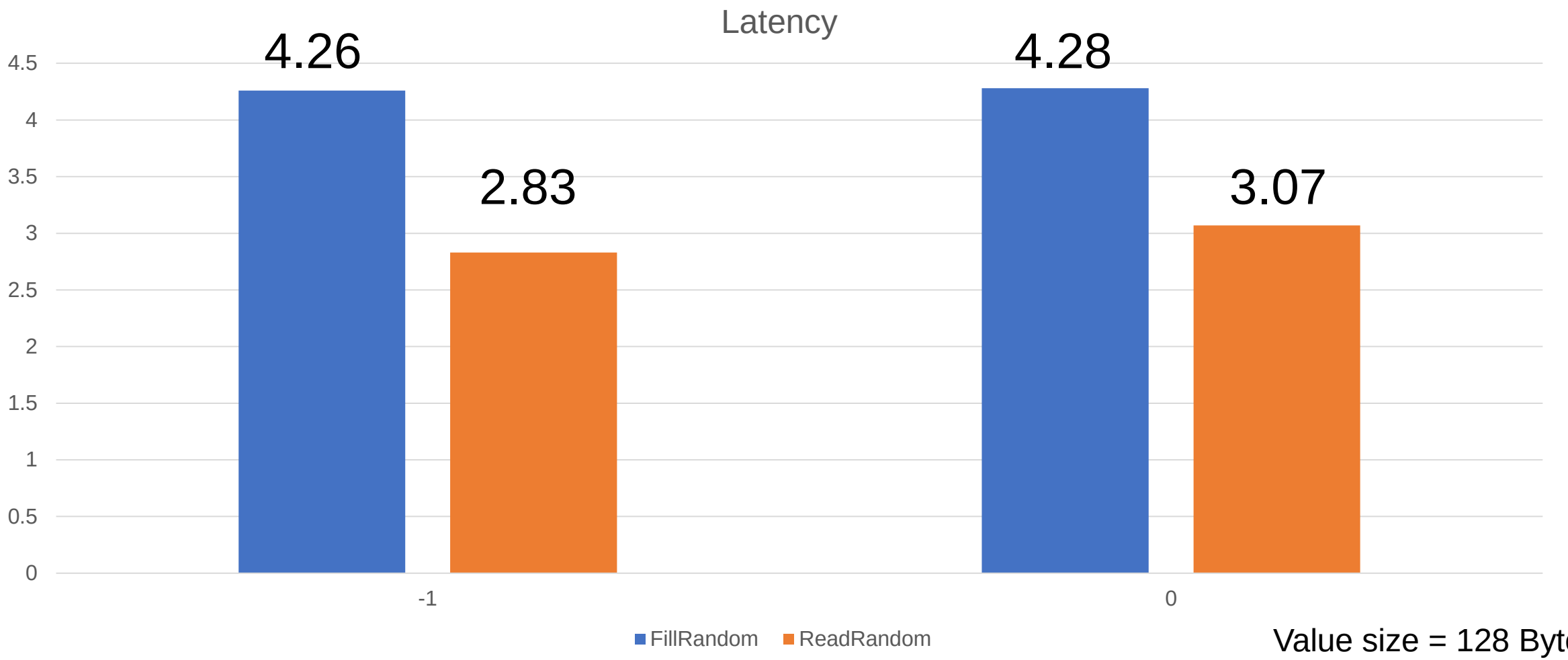
-Value_size가 128 Bytes일 때 블룸 필터를 사용하자 쓰기 시간은 7% 증가,
읽기시간은 7% 감소했다.

-Value_size가 1280 Bytes일 땐 쓰기 시간이 8% 증가, 읽기 시간이 9.5% 감소했다.

Conclusion

- 블룸 필터는 쓰기/읽기 시간에 비례한 성능 변화를 보인다.
- 읽기 시간이 쓰기 시간에 비해 상대적으로 길다면 블룸 필터를 사용하는 것이 좋을 것이고,
- 반대로 쓰기 시간이 더 길다면 블룸 필터를 사용하지 않는 것이 더 좋을 것이다.

Bloom bits -1 & 0



Value size = 128 Bytes

Bloom bits -1 & 0

	FillRandom	ReadRandom
Bloom bits = -1	51 (micros/op)	6.31
Bloom bits = 0	52.8	5.48

Value size = 1280 Bytes

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),
```

Bloom bits -1 & 0

```
  k_ = static_cast<size_t>(bits_per_key * 0.69);  // 0.69 =~ ln(2)
  if (k_ < 1) k_ = 1;
  if (k_ > 30) k_ = 30;
}


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;
```

Question