LevelDB Study
Bloom Filter

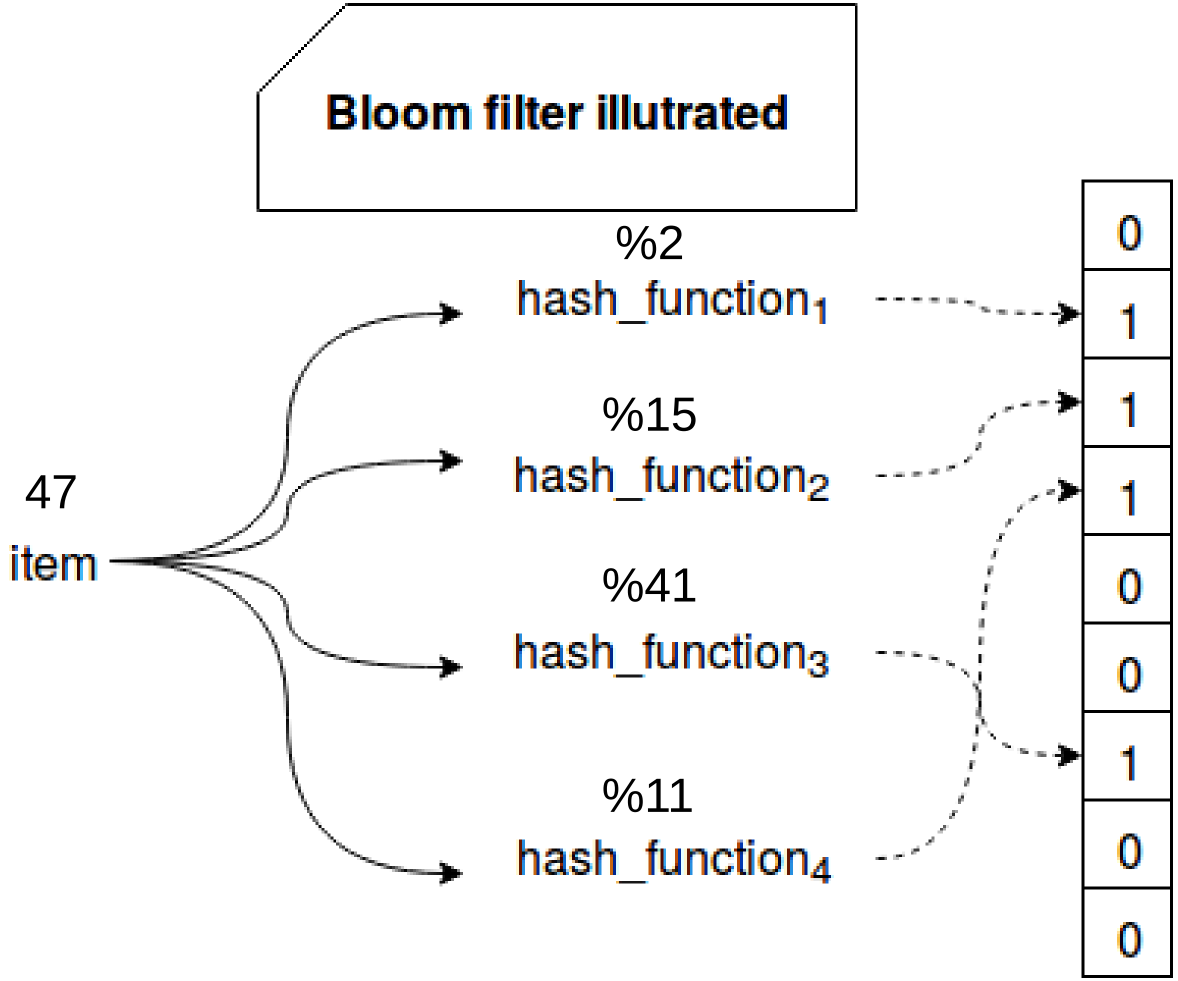
Made by Kim Han Su

E-Mail: khs20010327@naver.com

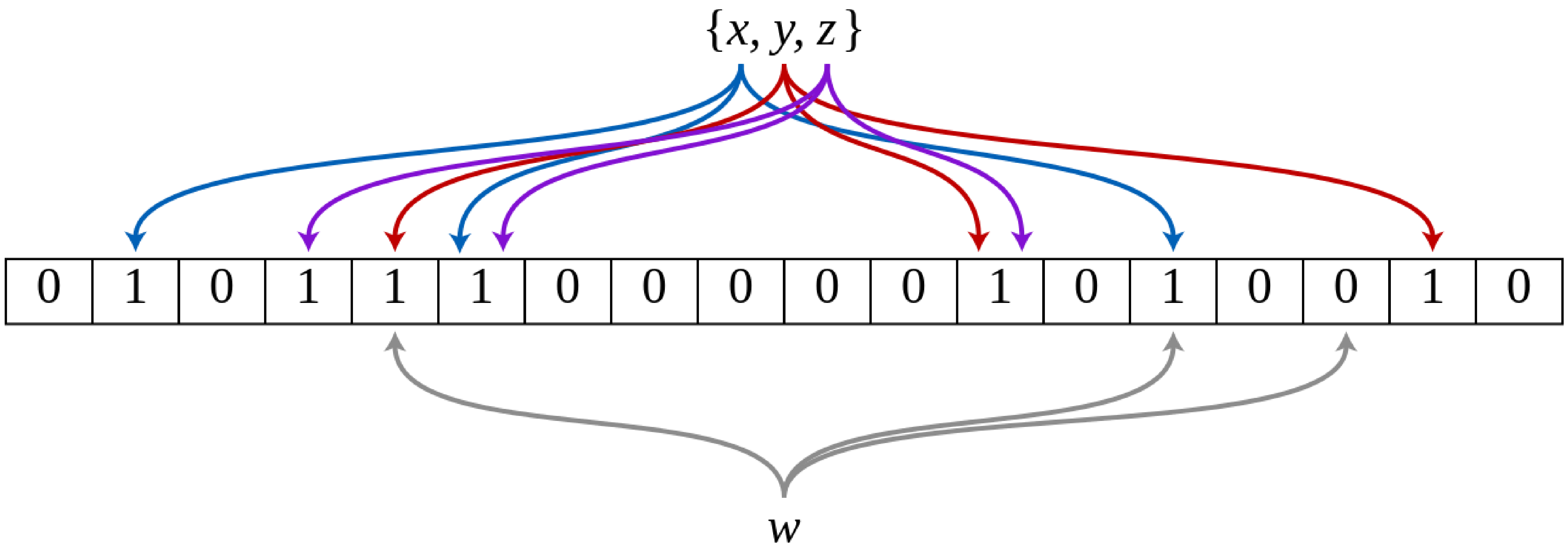
Contents

- Basic concept of Bloom Filter
- Hypothesis
- Measurement & Analysis
- Advanced concept about Hash function

Basic concept



Basic concept



Hypothesis

- LevelDB는 key당 10개의 bits를 default 값으로 사용한다.
- Bits per key의 값이 커질수록 Bloom Filter로 인한 성능이 향상되나,
- Bloom Filter를 처리하는데 필요한 부하도 같이 커지기에
- 최적의 Bits per key 값은 10일 것이다.

Basic concept

Hypothesis

Measurement & Analysis

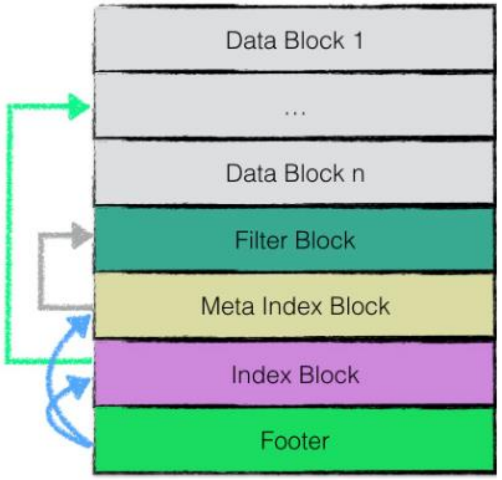
Advanced concept

- ✓ FillRandom
- ✓ ReadRandom

  \- The best number of hash functions

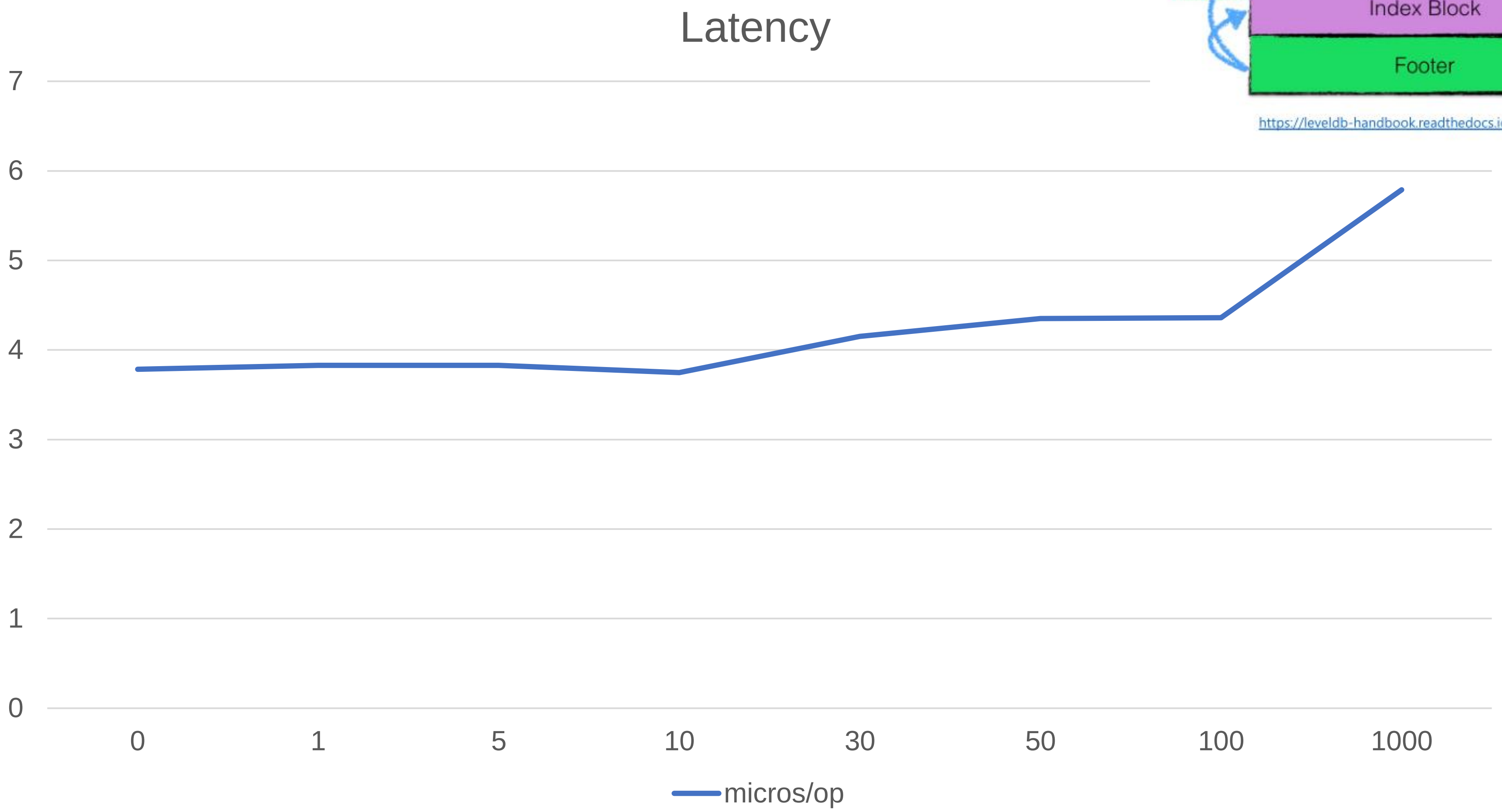
FillRandom

- Filter block
  - Bloom filter



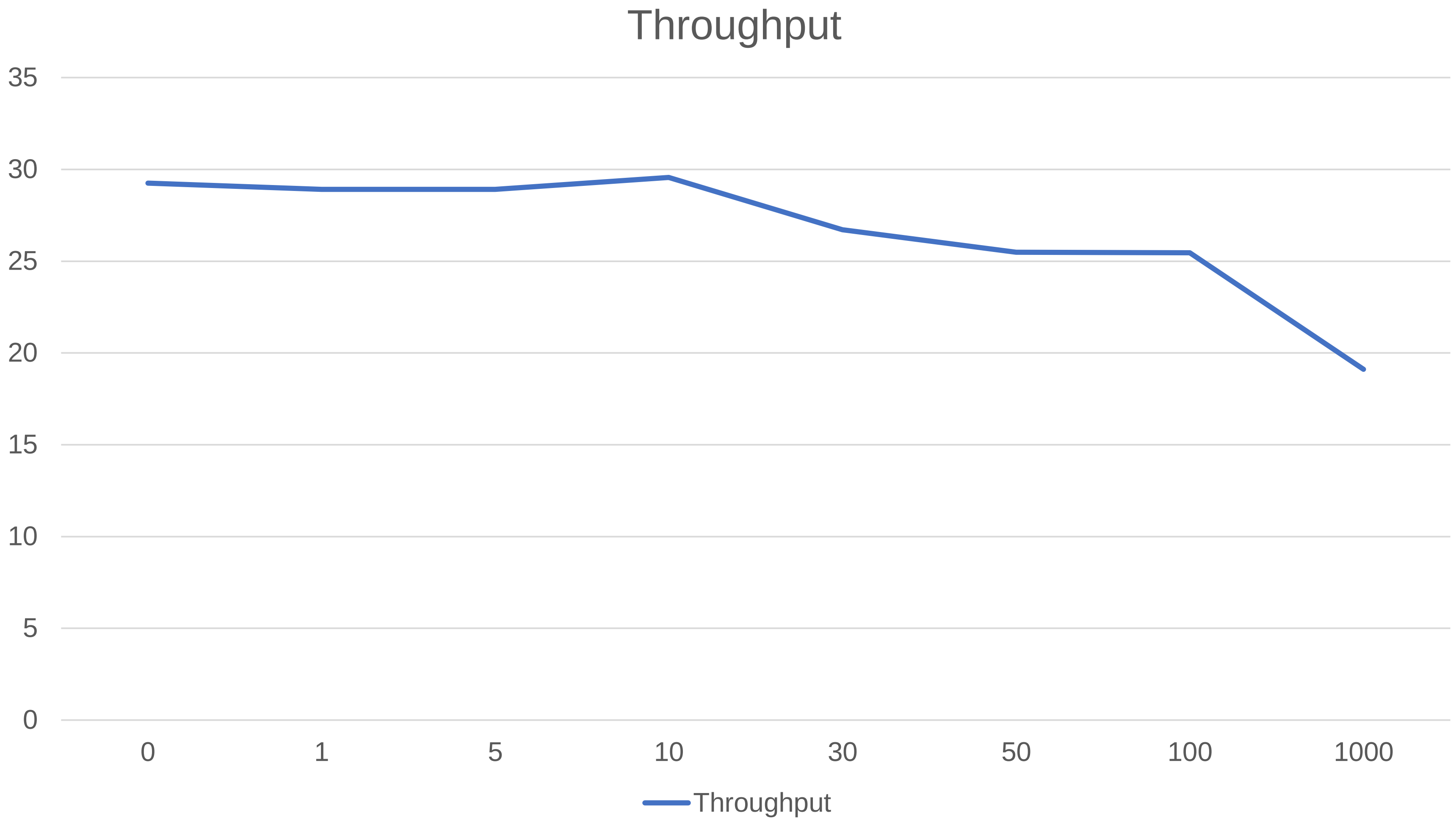
https://leveldb-handbook.readthedocs.io/zh/latest/

Bits per key	micros/op
0	3.7837
1	3.8289
5	3.8274
10	3.7458
30	4.1526
50	4.352
100	4.3591
1000	5.7891



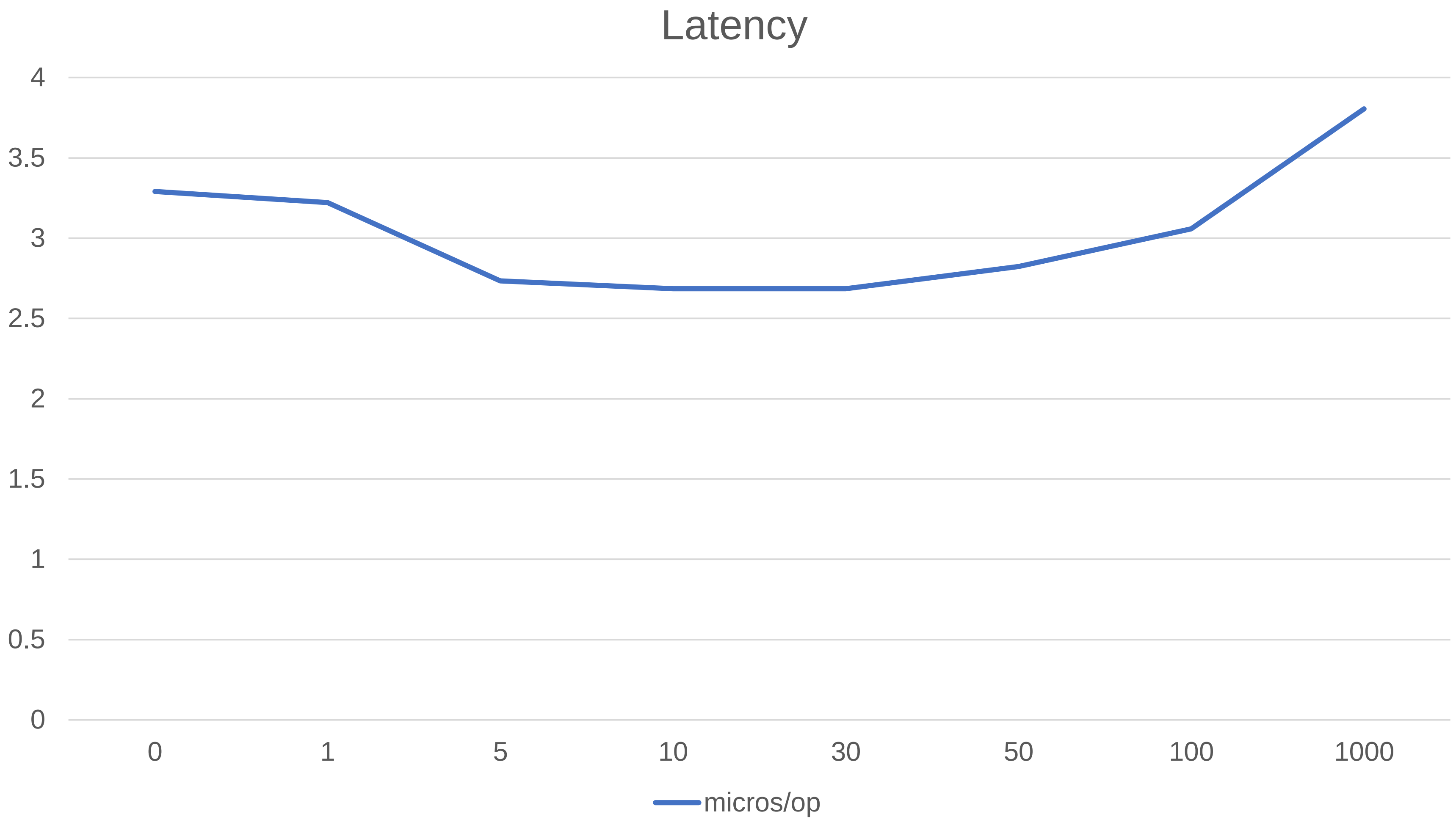
FillRandom

Bits per key	Throughput
0	29.26
1	28.91
5	28.92
10	29.56
30	26.72
50	25.49
100	25.46
1000	19.11

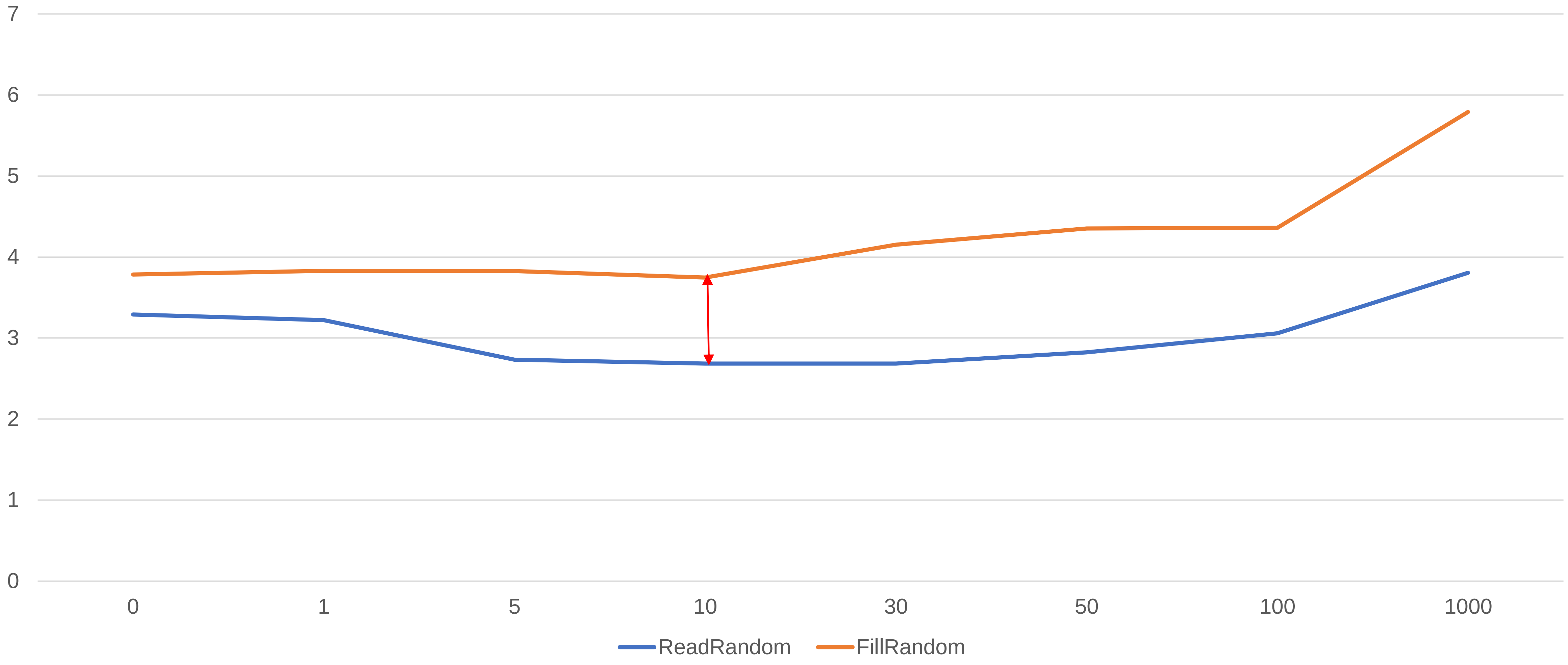


ReadRandom

Bits per key	micros/op
0	3.291
1	3.222
5	2.734
10	2.686
30	2.685
50	2.824
100	3.059
1000	3.805

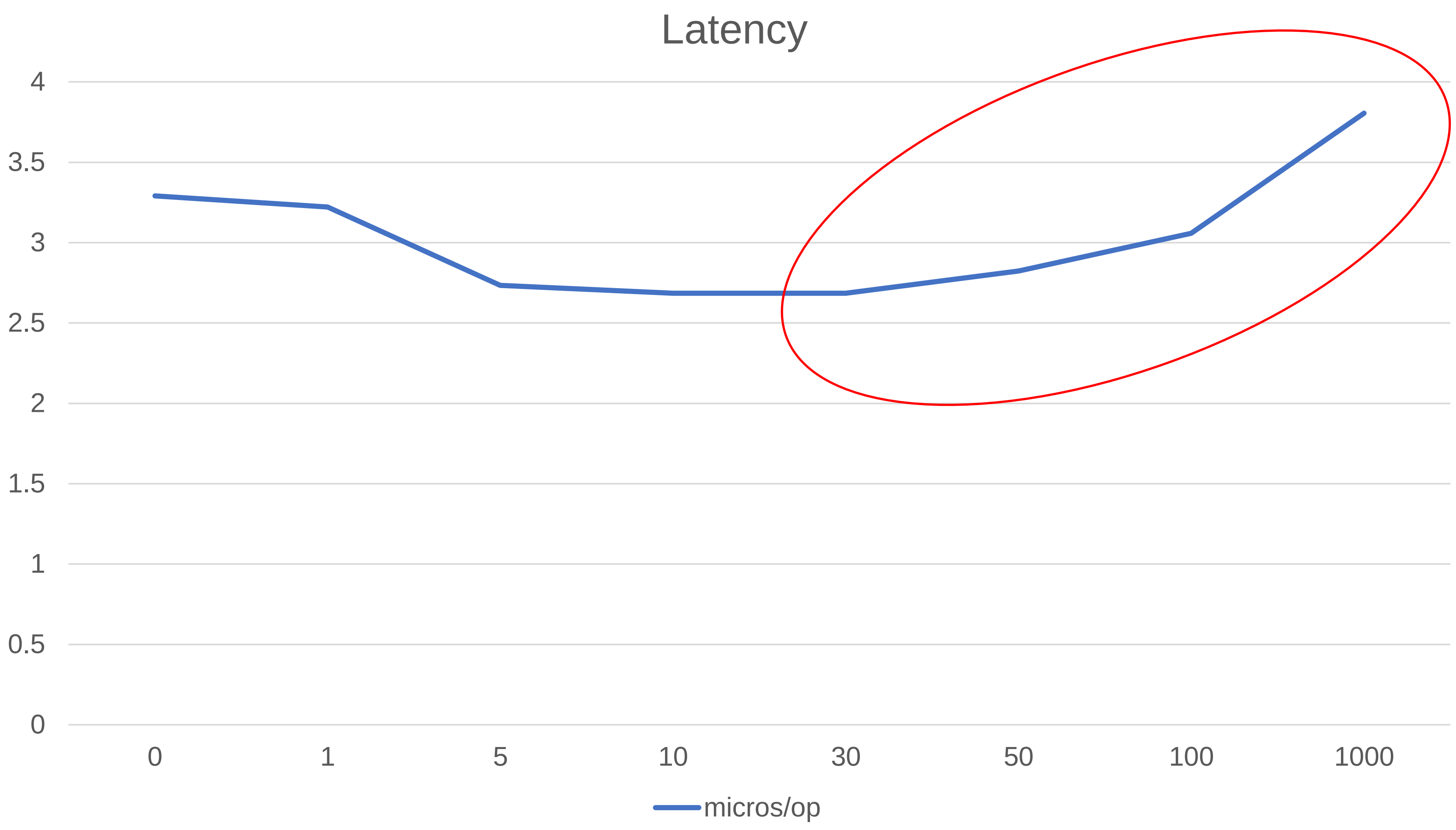


Latency



ReadRandom

Bits per key	micros/op
0	3.291
1	3.222
5	2.734
10	2.686
30	2.685
50	2.824
100	3.059
1000	3.805



0

3.292 3.317 3.218 3.186 3.362 3.275 3.270 3.421 3.260 3.299

1

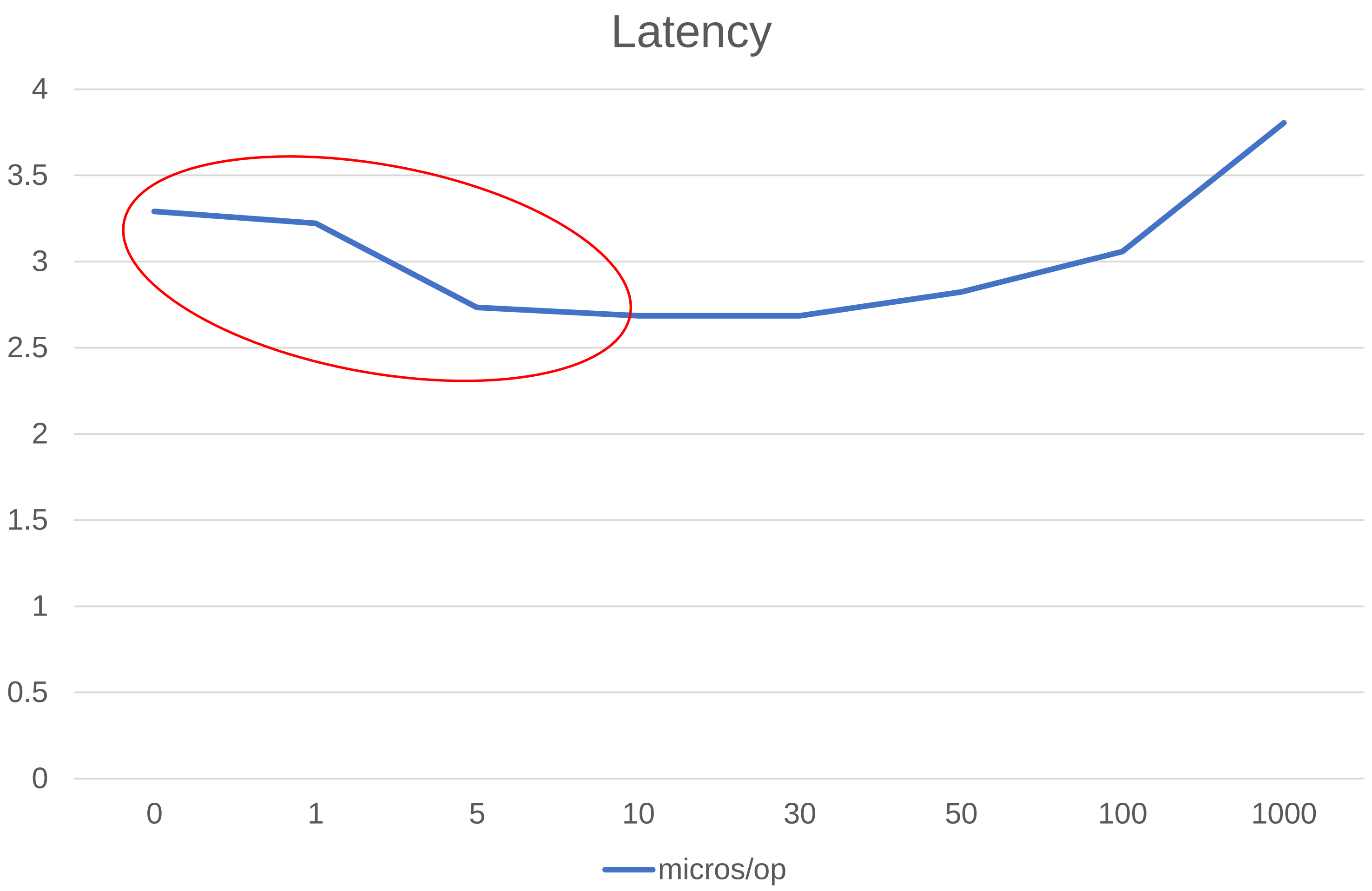
3.318 3.293 3.213 3.408 3.224 3.272 2.721 3.241 3.233 3.294

5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816



5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816

30

3.049 2.509 2.516 2.533 3.031 2.594 2.536 2.539 2.592 2.954

50

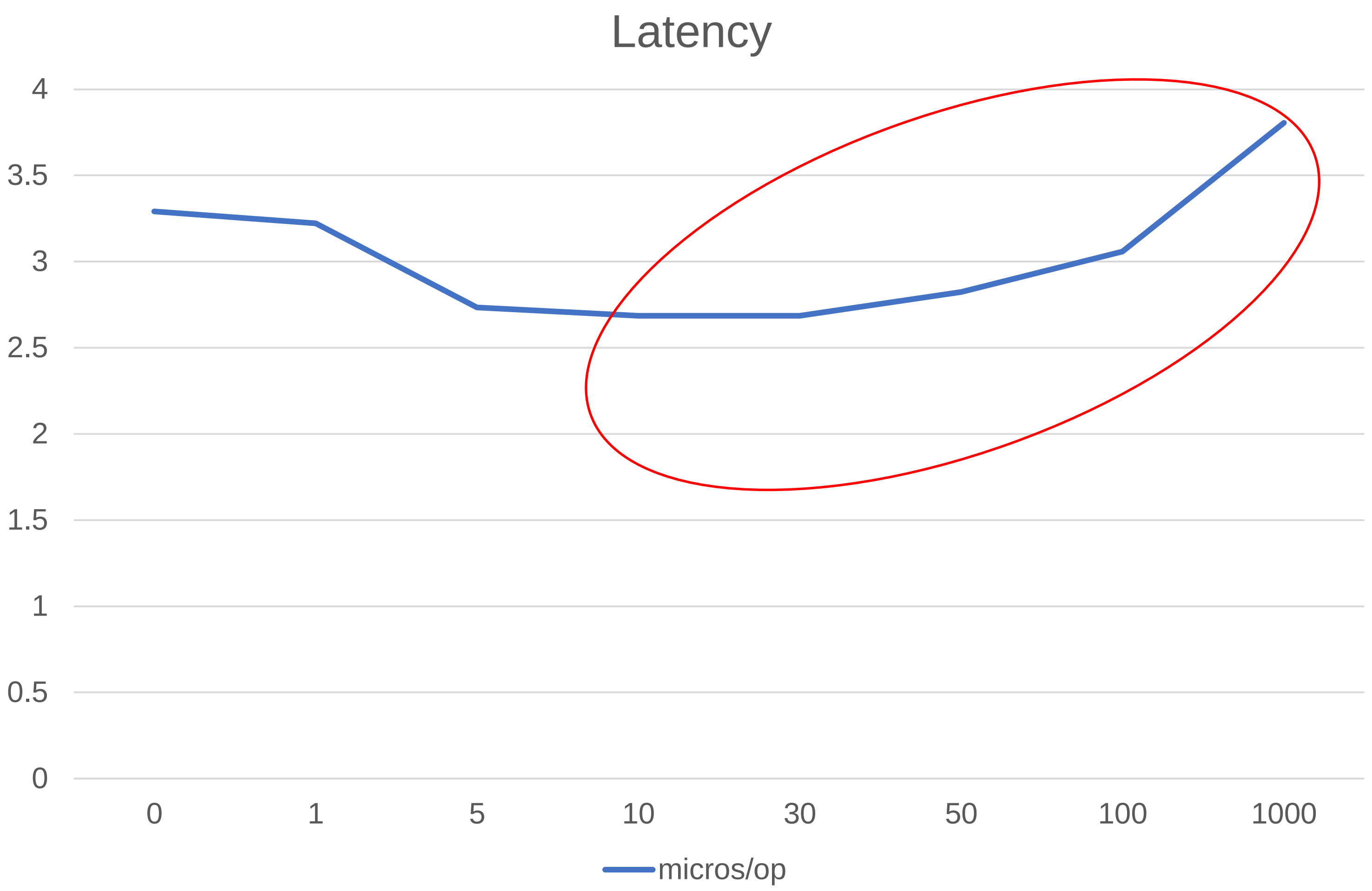
2.654 2.800 2.586 3.182 3.080 2.791 2.781 3.190 2.590 2.585

100

2.797 3.215 3.240 3.116 3.219 2.833 2.947 3.236 2.753 3.234

1000

4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061



0

3.292 3.317 3.218 3.186 3.362 3.275 3.270 3.421 3.260 3.299

1

3.318 3.293 3.213 3.408 3.224 3.272 2.721 3.241 3.233 3.294

5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816

30

3.049 2.509 2.516 2.533 3.031 2.594 2.536 2.539 2.592 2.954

50

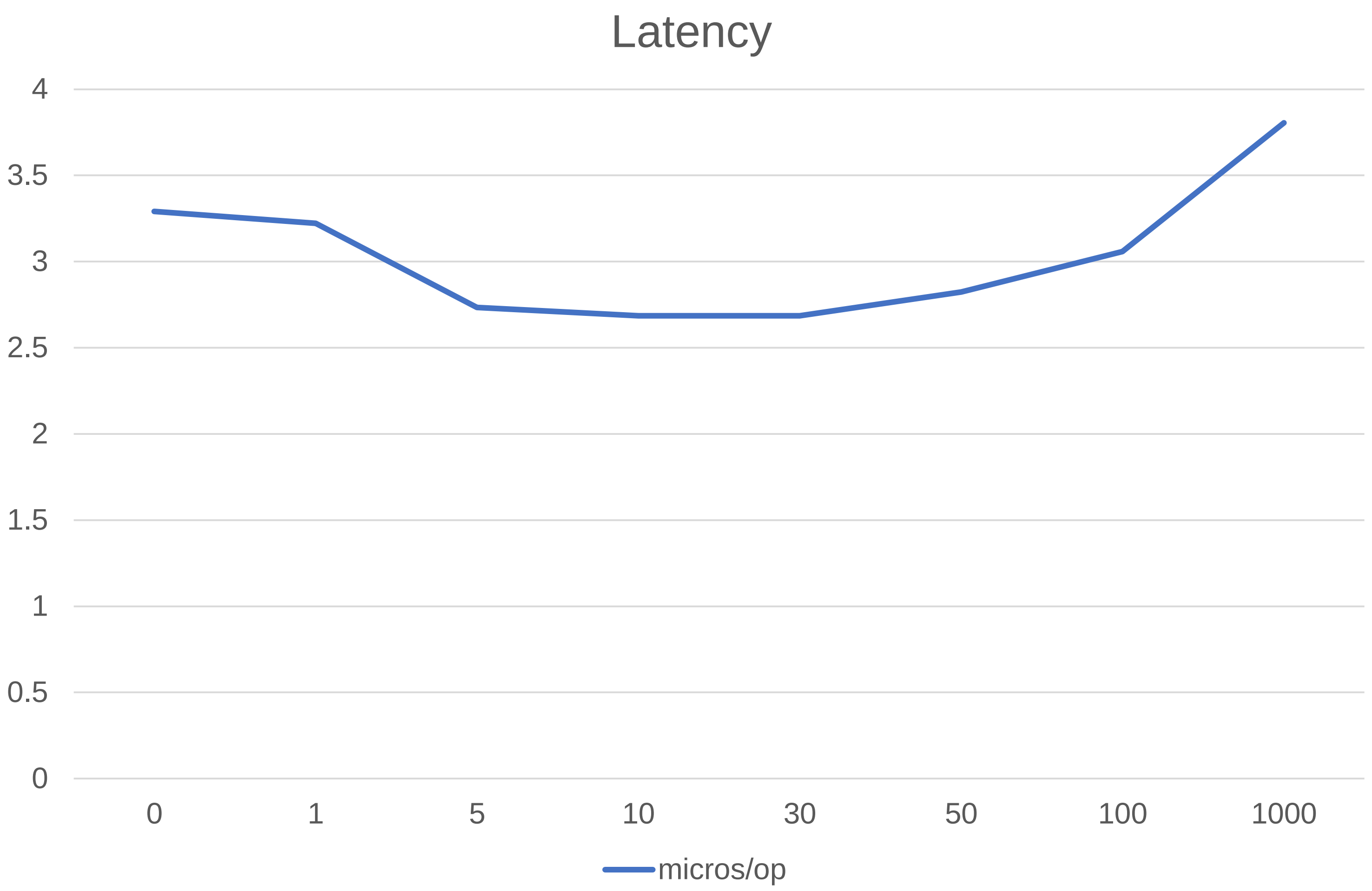
2.654 2.800 2.586 3.182 3.080 2.791 2.781 3.190 2.590 2.585

100

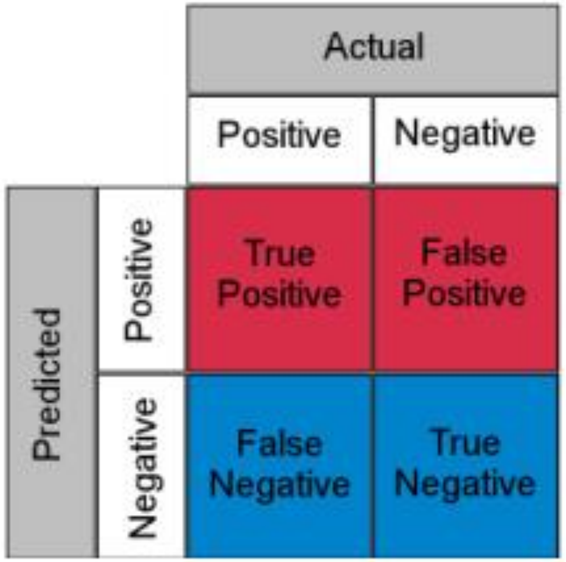
2.797 3.215 3.240 3.116 3.219 2.833 2.947 3.236 2.753 3.234

1000

4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061



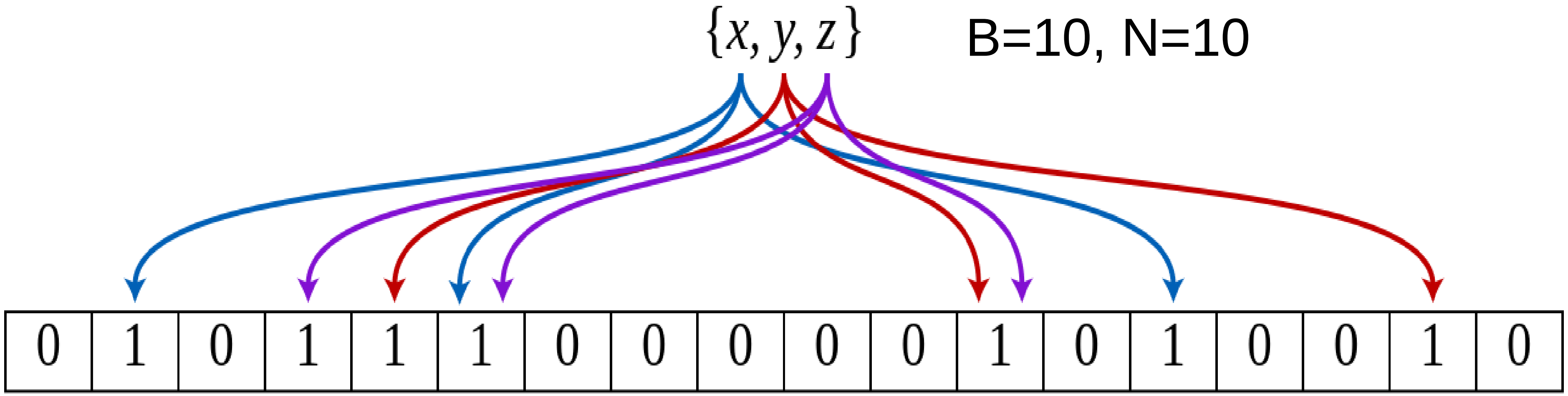
Bloom Filter



https://commons.wikimedia.org/wiki/File:ConfusionMatrixRedBlue.png

- Good property: No false negative
- Issue: can yield false positives

- Bits per Key = B, Num = N, Bloom filter bit array’s capacity = M

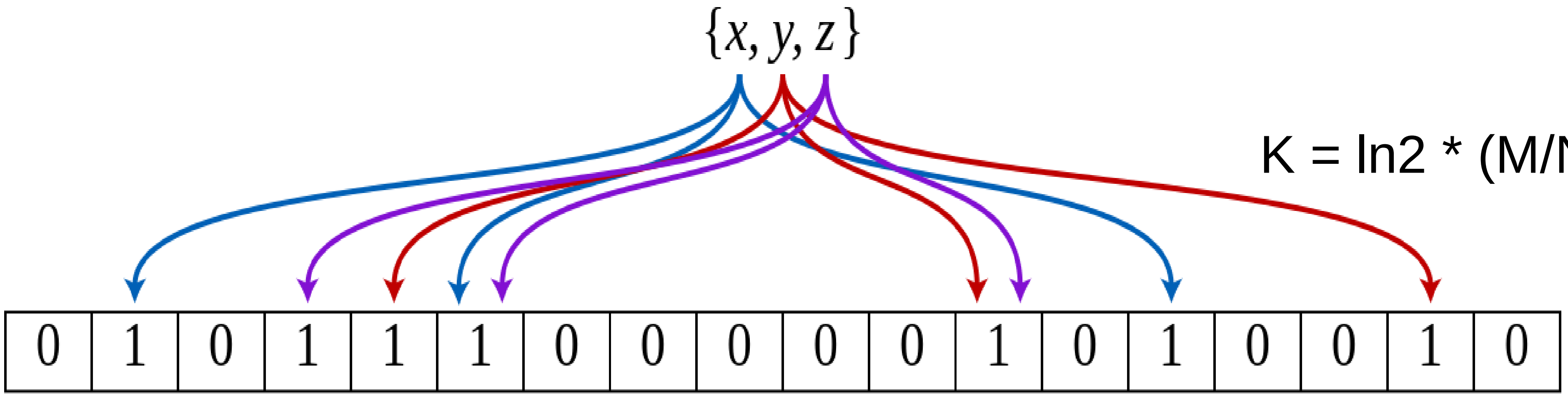


B=10, N=10

M = B * N = 100

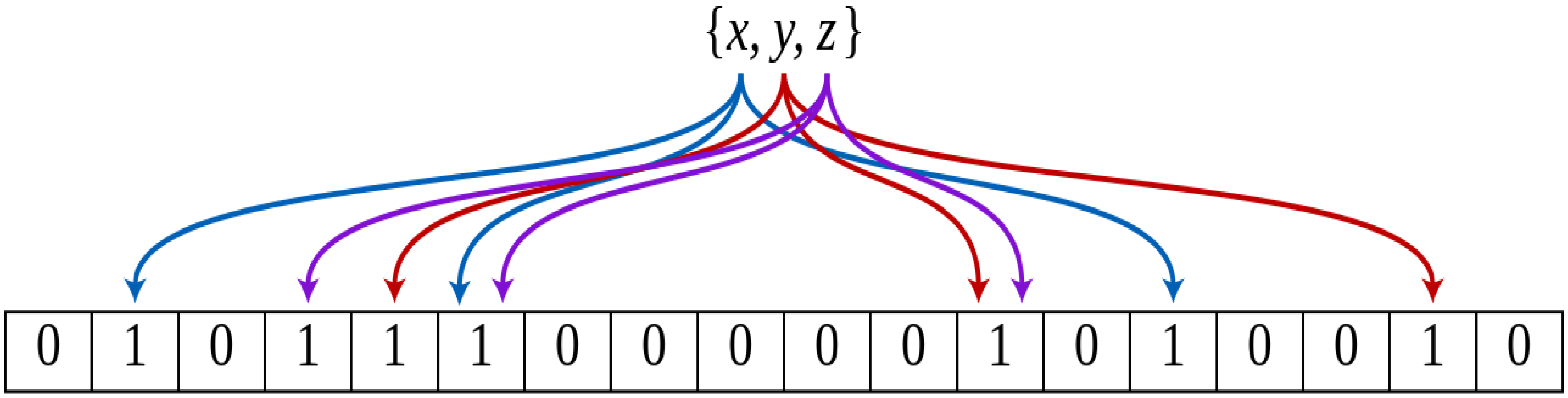
```
void CreateFilter(const Slice* keys, int n, std::string* dst) const override {
  // Compute bloom filter size (in both bits and bytes)
  size_t bits = n * bits_per_key_;
```

- Bits per Key = B, Num = N, Bloom filter bit array’s capacity = M
- The number of Hash functions = K



K = ln2 * (M/N) = ln2 * B

```
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)
```

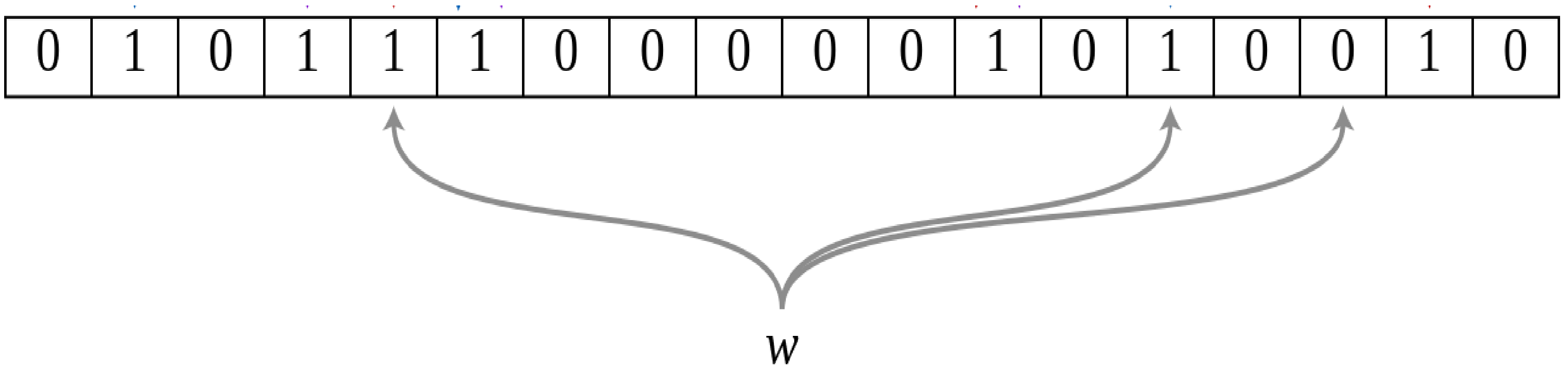


1 Key, 1 Hash, Bit 1 = 1/M

1 Key, 1 Hash, Bit 0 = 1 - 1/M

N Key K Hash, Bit 0 = $\left(1 - \frac{1}{m}\right)^{kn}$

N Key K Hash, Bit 1 = $1 - \left(1 - \frac{1}{m}\right)^{kn}$



N Key K Hash, Bit 1 = $1-\left(1-\frac{1}{m}\right)^{kn}$

N Key K Hash, K Bit 1 = $\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} = P_{false\ postive}$

$$P_{false\,postive} = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

$$\lim_{m\to\infty}\left(1 - \frac{1}{m}\right)^{m} = \frac{1}{e} \longrightarrow \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1 - e^{\frac{-kn}{m}}\right)^{k}$$

- $k = \frac{m}{n}\ln 2$ 일 때 $P_{false\,postive}$ 는 최솟값을 지닌다.
- $k = \frac{m}{n}\ln 2$ 일 때 k 값이 클수록 $P_{false\,postive}$ 값이 작아진다.

-> k = ln2 * B 이므로, Bits per key 값이 커질수록 $P_{false\,postive}$ 값이 작아진다

0

3.292 3.317 3.218 3.186 3.362 3.275 3.270 3.421 3.260 3.299

1

3.318 3.293 3.213 3.408 3.224 3.272 2.721 3.241 3.233 3.294

5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816

30

3.049 2.509 2.516 2.533 3.031 2.594 2.536 2.539 2.592 2.954

50

2.654 2.800 2.586 3.182 3.080 2.791 2.781 3.190 2.590 2.585

100

2.797 3.215 3.240 3.116 3.219 2.833 2.947 3.236 2.753 3.234

1000

4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061

- $k = \frac{m}{n} \ln 2$ 일 때 k 값이 클수록 $P_{false\ postive}$ 값이 작아진다.

-> k = ln2 * B 이므로, Bits per key 값이 커질수록 $P_{false\ postive}$ 값이 작아진다

0

3.292 3.317 3.218 3.186 3.362 3.275 3.270 3.421 3.260 3.299

1

3.318 3.293 3.213 3.408 3.224 3.272 2.721 3.241 3.233 3.294

5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816

30

3.049 2.509 2.516 2.533 3.031 2.594 2.536 2.539 2.592 2.954

50

2.654 2.800 2.586 3.182 3.080 2.791 2.781 3.190 2.590 2.585

100

2.797 3.215 3.240 3.116 3.219 2.833 2.947 3.236 2.753 3.234

1000

4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061

- $k = \frac{m}{n} \ln 2$ 일 때 k 값이 클수록 $P_{false\ postive}$ 값이 작아진다.

-> k = ln2 * B 이므로, Bits per key 값이 커질수록 $P_{false\ postive}$ 값이 작아진다

Leveldb/util/bloom.cc

```
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;
  }
```

Leveldb/util/bloom.cc

```
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;

  }
```

DANKOOK UNIVERSITY

Dankook University System Software Laboratory

Result

1000 bits per key

FillRandom 5.885 5.845 5.695 5.764 5.603 5.776 5.732 5.720 5.949 5.922

Throughput 18.8 18.9 19.4 19.2 19.7 19.2 19.3 19.3 18.6 18.7

ReadRandom 4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061

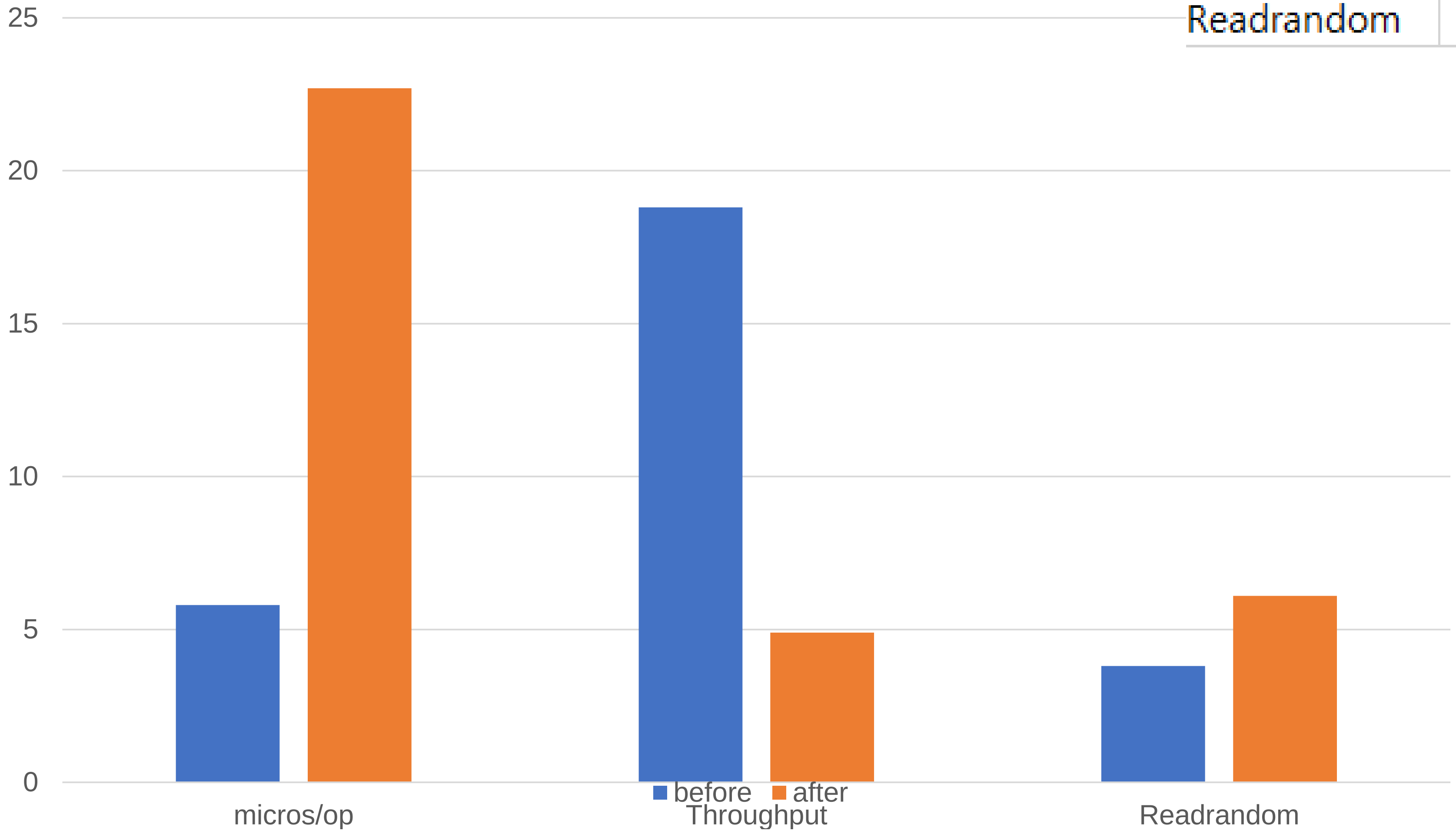
FillRandom 22.72 22.74 22.63 22.77 22.60 22.58 22.77 22.58 22.66 22.661

Throughput 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9

ReadRandom 6.086 5.981 6.016 6.017 5.994 6.093 6.017 5.898 5.985 6.275

Result

	before	after
micros/op	5.8	22.7
Throughput	18.8	4.9
Readrandom	3.8	6.1



Result

1000 bits per key

FillRandom 5.885 5.845 5.695 5.764 5.603 5.776 5.732 5.720 5.949 5.922

Throughput 18.8 18.9 19.4 19.2 19.7 19.2 19.3 19.3 18.6 18.7

ReadRandom 4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061

FillRandom 22.72 22.74 22.63 22.77 22.60 22.58 22.77 22.58 22.66 22.661

Throughput 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9

ReadRandom 6.086 5.981 6.016 6.017 5.994 6.093 6.017 5.898 5.985 6.275

Conclusion

- 1.Bloom Filter를 사용하면 전체적인 성능이 향상된다.

- 2.성능이 특히 더 많이 향상되는 경우가 존재하며,
이는 False Positive가 덜 발생하기 때문으로 추정된다.

- 3.False Positive는 Hash 함수의 개수가 ln2 * bits per key일 때,
그리고 bits per key 값이 커질수록 발생 확률이 작아진다.

- 4.단 bits per key가 커질수록 이를 처리하는데 필요한 연산이
증가하기에 전체적인 성능은 되려 떨어진다.

0

3.292 3.317 3.218 3.186 3.362 3.275 3.270 3.421 3.260 3.299

1

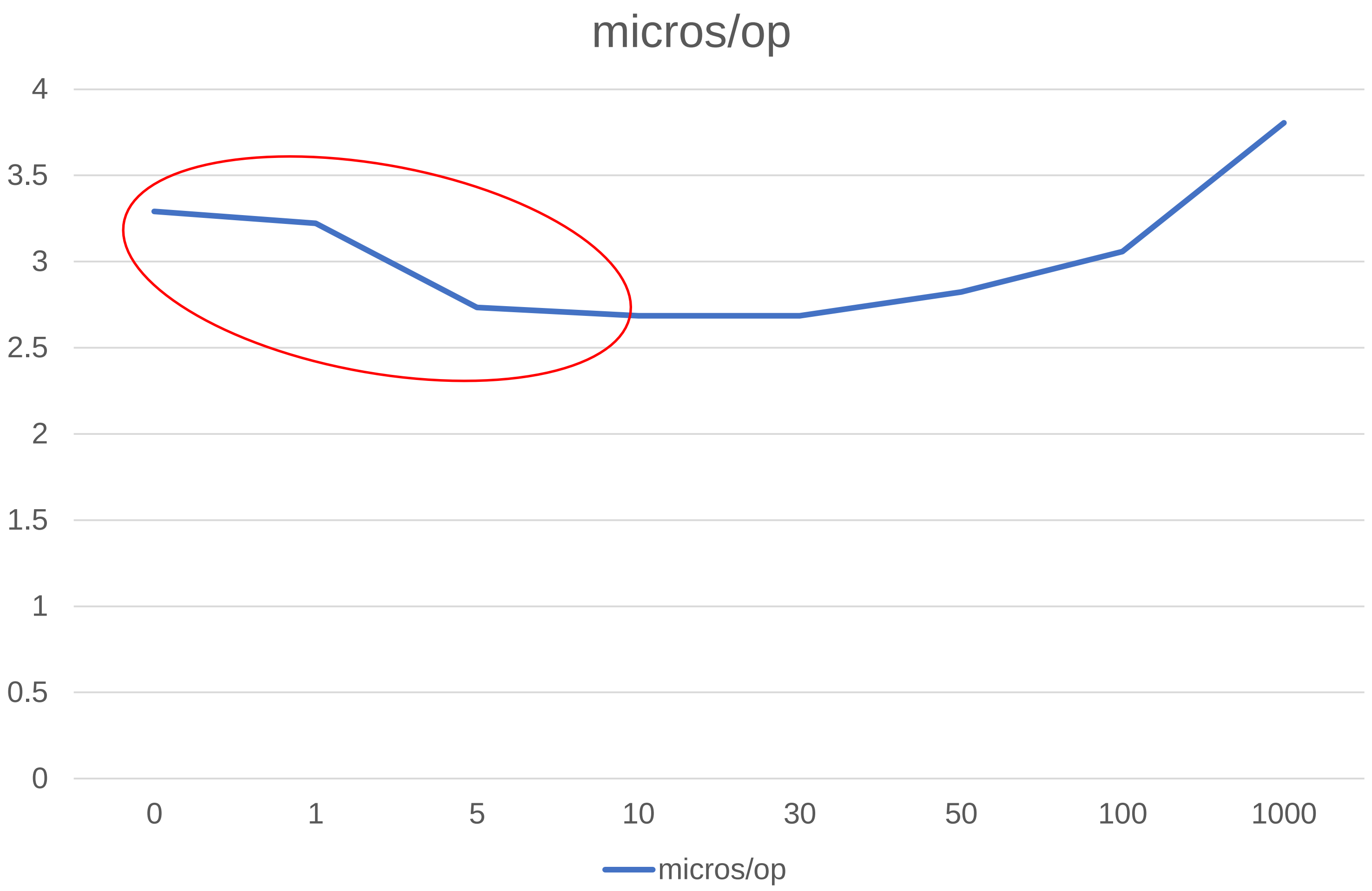
3.318 3.293 3.213 3.408 3.224 3.272 2.721 3.241 3.233 3.294

5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816



5

2.368 2.419 2.776 2.963 2.809 2.843 2.855 2.925 2.411 2.974

10

2.402 2.356 2.816 2.794 2.937 2.766 2.805 2.330 2.837 2.816

30

3.049 2.509 2.516 2.533 3.031 2.594 2.536 2.539 2.592 2.954

50

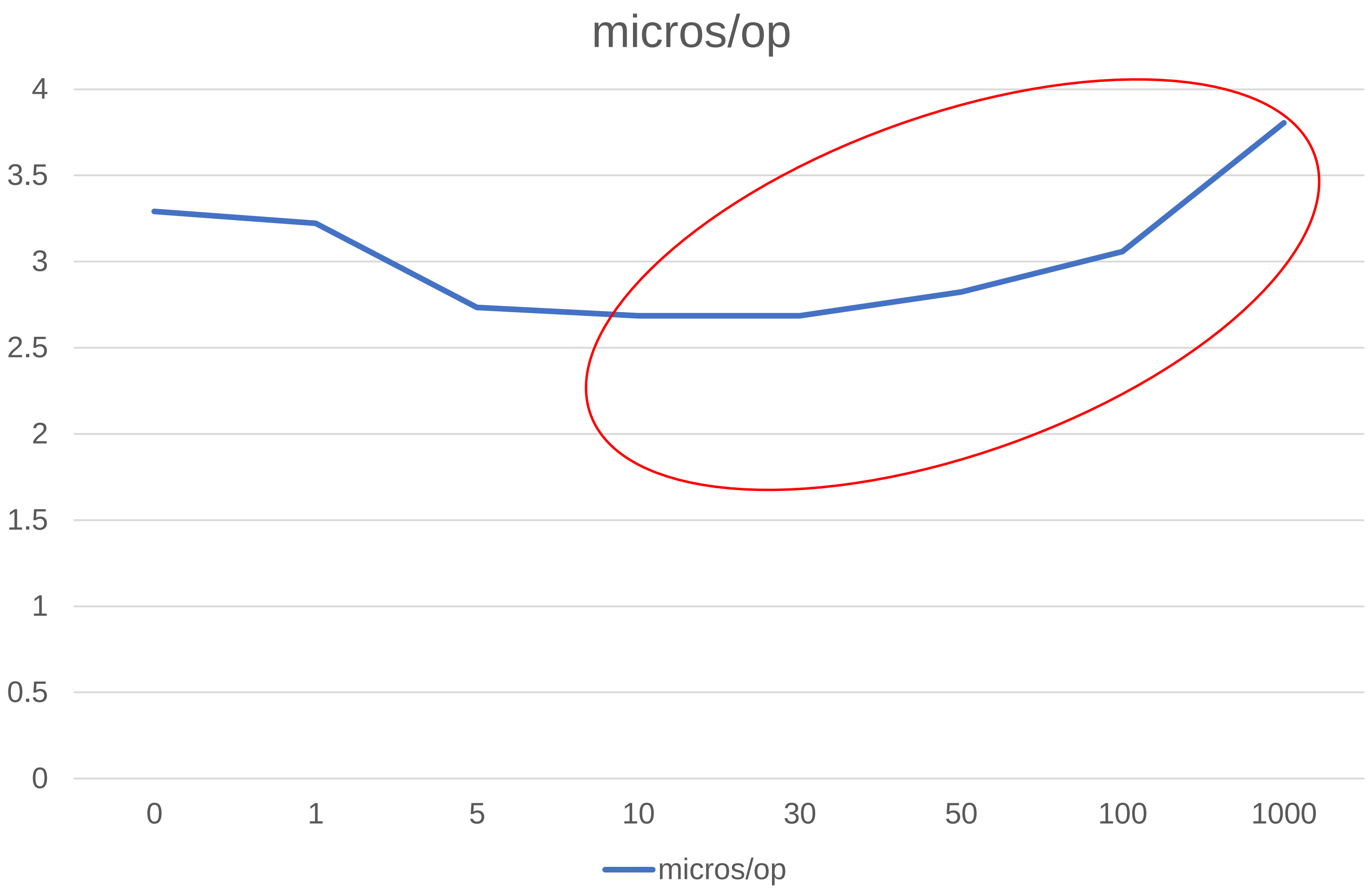
2.654 2.800 2.586 3.182 3.080 2.791 2.781 3.190 2.590 2.585

100

2.797 3.215 3.240 3.116 3.219 2.833 2.947 3.236 2.753 3.234

1000

4.028 4.134 3.397 3.423 4.192 3.375 3.415 4.196 3.838 4.061



Reference

- leveldb/util/bloom.cc
- https://github.com/google/leveldb/blob/main/util/bloom.cc

- 확률적 자료구조를 이용한 추정
- https://d2.naver.com/helloworld/749531

QnA