ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores

Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, Yinlong Xu

Contact: ykli@ustc.edu.cn

University of Science and Technology of China

Outline

- Background
  - Key-value (KV) stores and LSM tree
- Motivation
  - Read amplification problem in KV stores
- Design of ElasticBF
- Performance Evaluation
- Conclusion

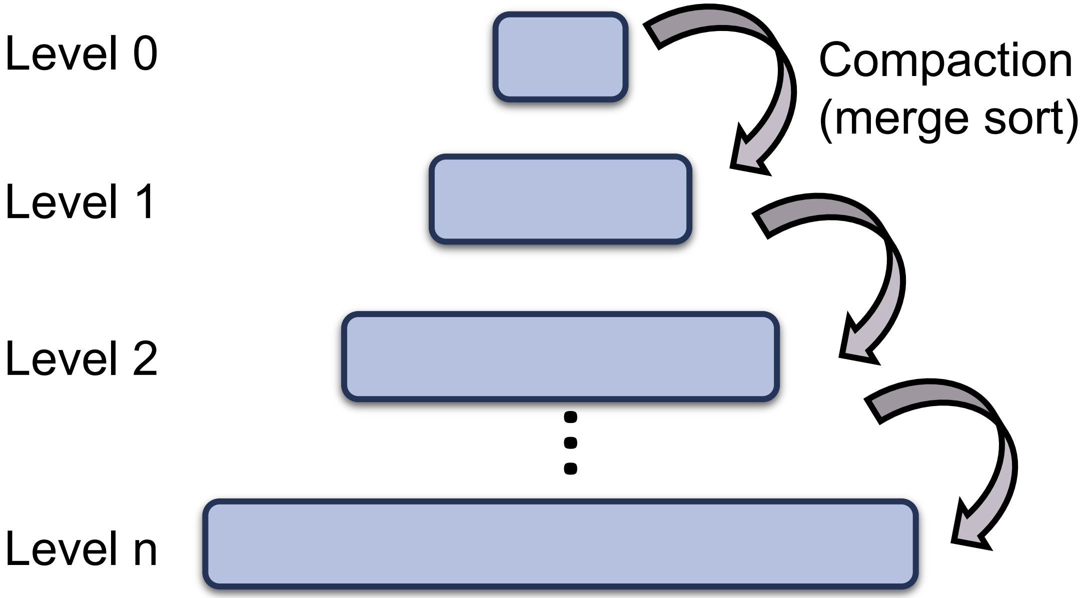
Background

- Key-value (KV) store has become an important storage engine for many applications
  - Cloud systems
  - Social networks
  - …

- Examples of KV stores
  - Hbase @ Apache
  - LevelDB @ Google
  - RocksDB @ Facebook
  - …

LSM Tree

- The most common design of KV stores is based on LSM-tree (log structured merge tree)



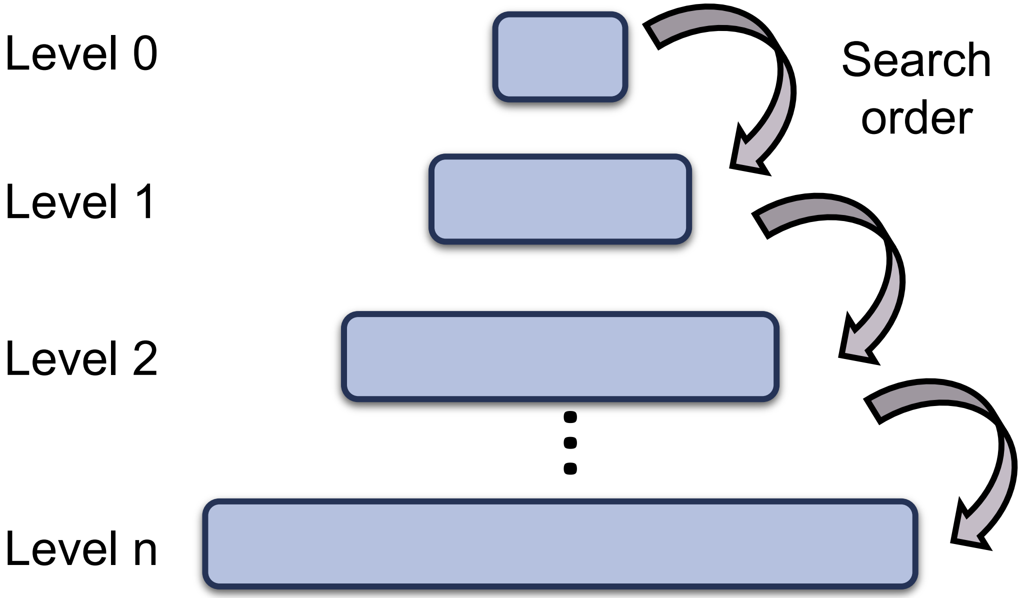
Fully sorted in each level

Compaction incurs **write amplification**

Data is written to Level 0 first, then merged to Level 1 via compaction, then Level 2, and so on.

LSM Tree

- The most common design of KV stores is based on LSM-tree (log structured merge tree)



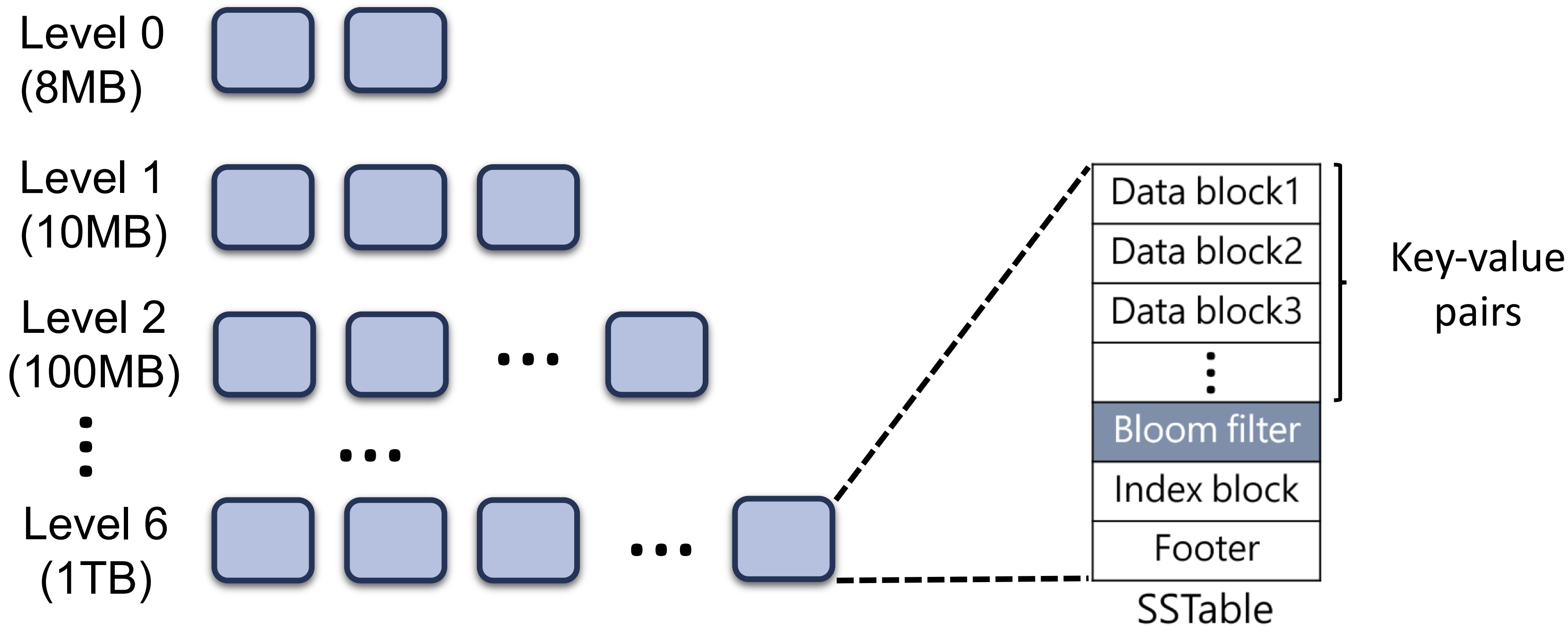
Fully sorted in each level

Searching multiple levels is needed

Looking up a key requires multiple I/O requests as it may require to search in multiple levels (read amplification).

LevelDB

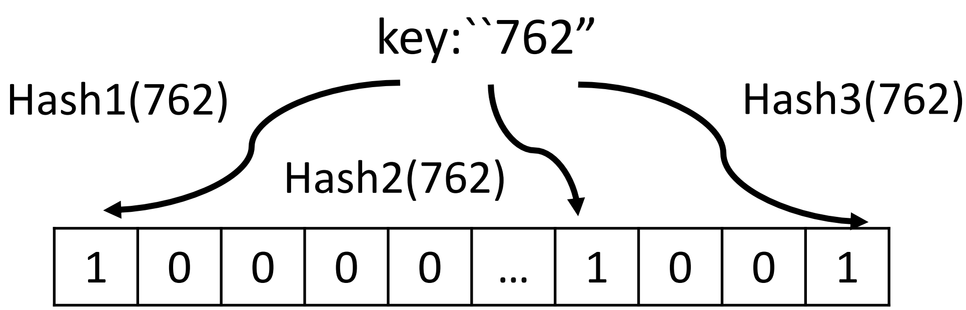
- One typical implementation of LSM tree
  - Focus on data layout on disk



It suffers from read amplification problem, especially for a large KV store which has multiple levels.

Bloom Filter

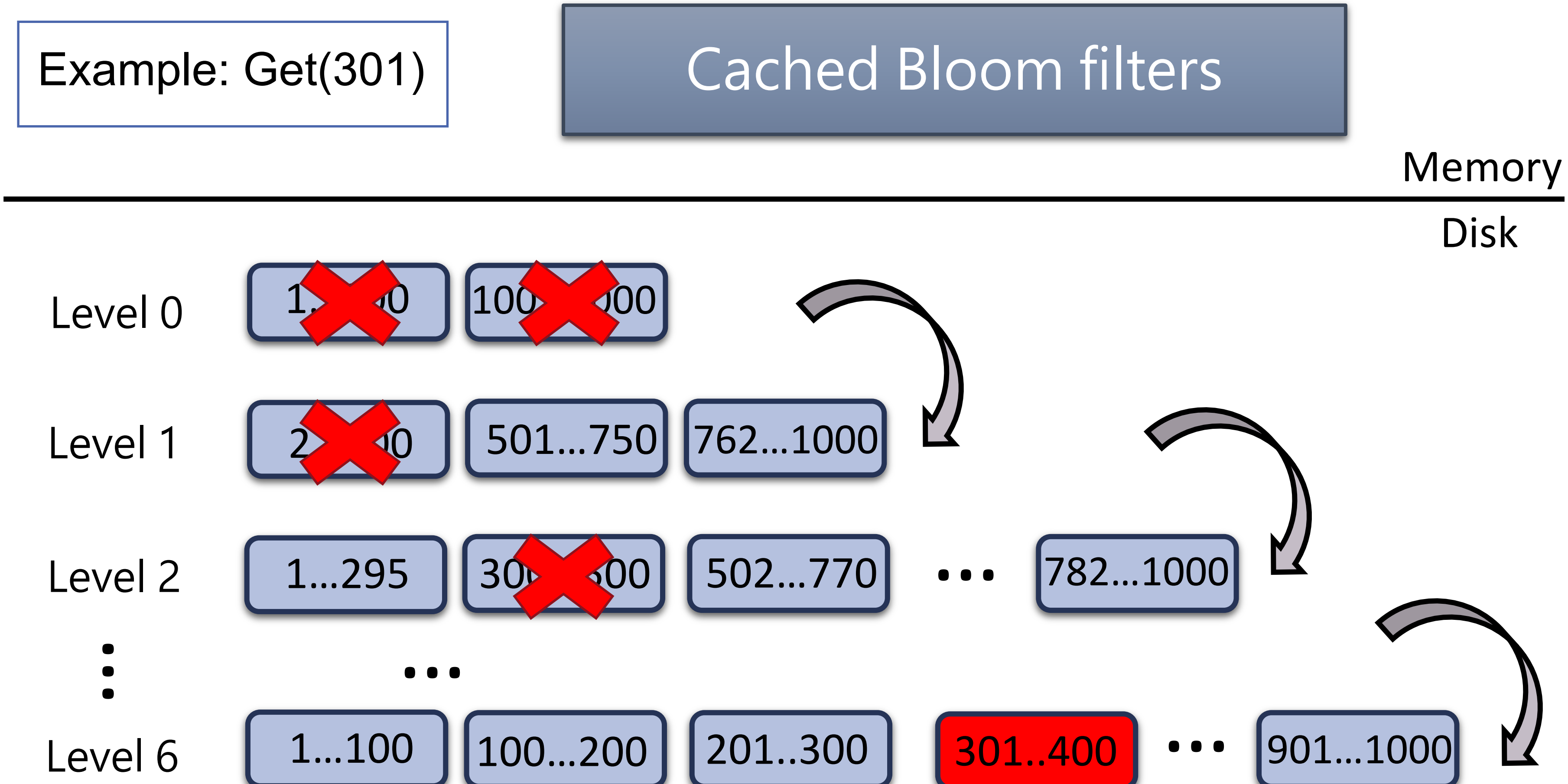
- Bloom filter in each SSTable
  - A bit array with multiple hash functions
  - Help quickly identify whether a key exists in an SSTable or not



  - Bloom filter suffers from false positive (hash collision)
    - False positive rate (FPR) : 0.6185^{b} (b: Bits-per-key)

Bits-per-key	2bits	3bits	4bits	5bits	6bits
FPR	40%	23.7%	14.7%	9.2%	5.6%

Read Flow with Bloom Filter



Bloom filters are required to be cached in memory

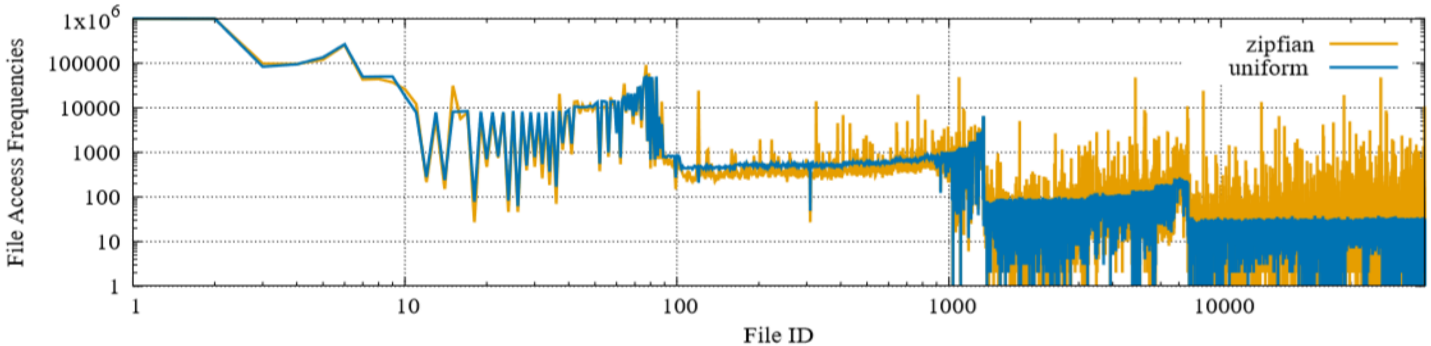
Motivation

- KV stores suffer from large read amplification
  - Bloom filter reduces read I/O, but has false positive
  - Reducing false positive may need to allocate many bits for each key, incurs large memory overhead

- Question: how to improve the Bloom filter design with limited memory consumption so as to
  - reduce extra I/O requests and
  - improve read performance of KV stores

Main Idea

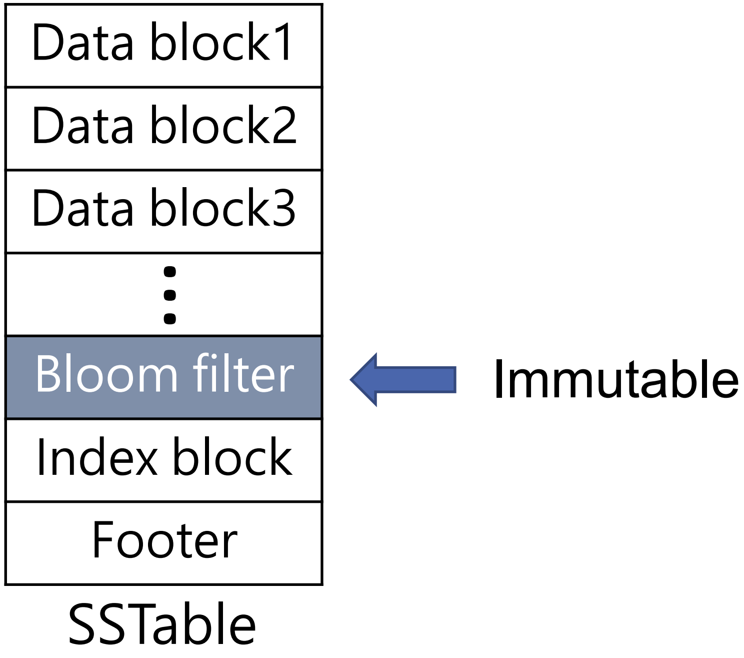
- Observation
  - Access frequencies of SSTables in low levels are higher
  - Unevenness of accesses even within the same level



- Main idea: ElasticBF
  - An elastic scheme according to access frequency
  - SSTables with high (low) access frequency
    - More (less) powerful Bloom filter (i.e., more (fewer) bits per key)
    - Lower (higher) false positive rate: fewer extra I/Os
    - Larger (smaller) memory consumption

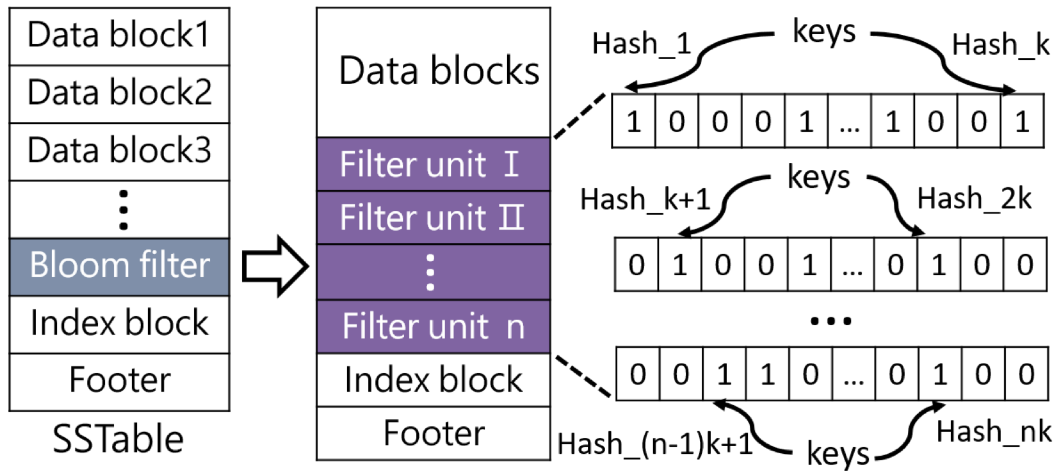
ElasticBF Design

- Challenge to realize an elastic scheme according to access frequency of SSTables
  - Data organization in SSTable is fixed after creation
  - Adjusting the Bloom filter in SSTables requires to reorganize the data



ElasticBF Design

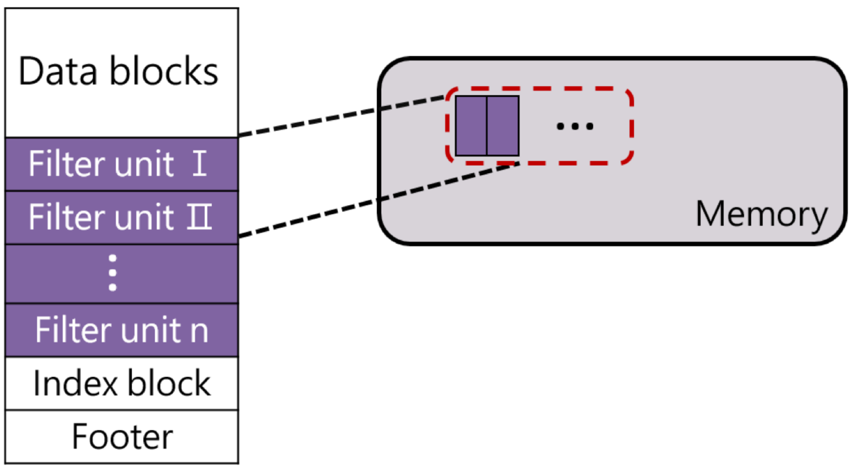
- Choice of ElasticBF: Step 1
  - Build multiple small filter units in each SSTable with different and independent hash functions



- Rationale
  - **Separability**: Multiple filters have the same FPR as a single filter with the same *bits-per-key*
    - FRR of n filter units : $\prod_{i=1}^{n} 0.6185^{b_i} = 0.6185^{b}$ ($\sum_{i=1}^{n} b_i = b$)

ElasticBF Design

- Choice of ElasticBF: Step 2
  - Dynamically adjust the filter units in memory for each SSTable according to its access frequency
    - Enable more filter units by loading them into memory
    - Disable in-memory filter units by simply discarding them



Elastic feature: false positive rate can be dynamically adjusted

Data organization in SSTable does not change

Key Issues

How to determine the most appropriate number of filter units for each SSTable ?

Adjusting Rule

How to realize a dynamic adjustment with small overhead ?

Multi-Queue

Adjusting Rule

- **Goal**
  - Try to reduce the extra I/Os caused by false positive

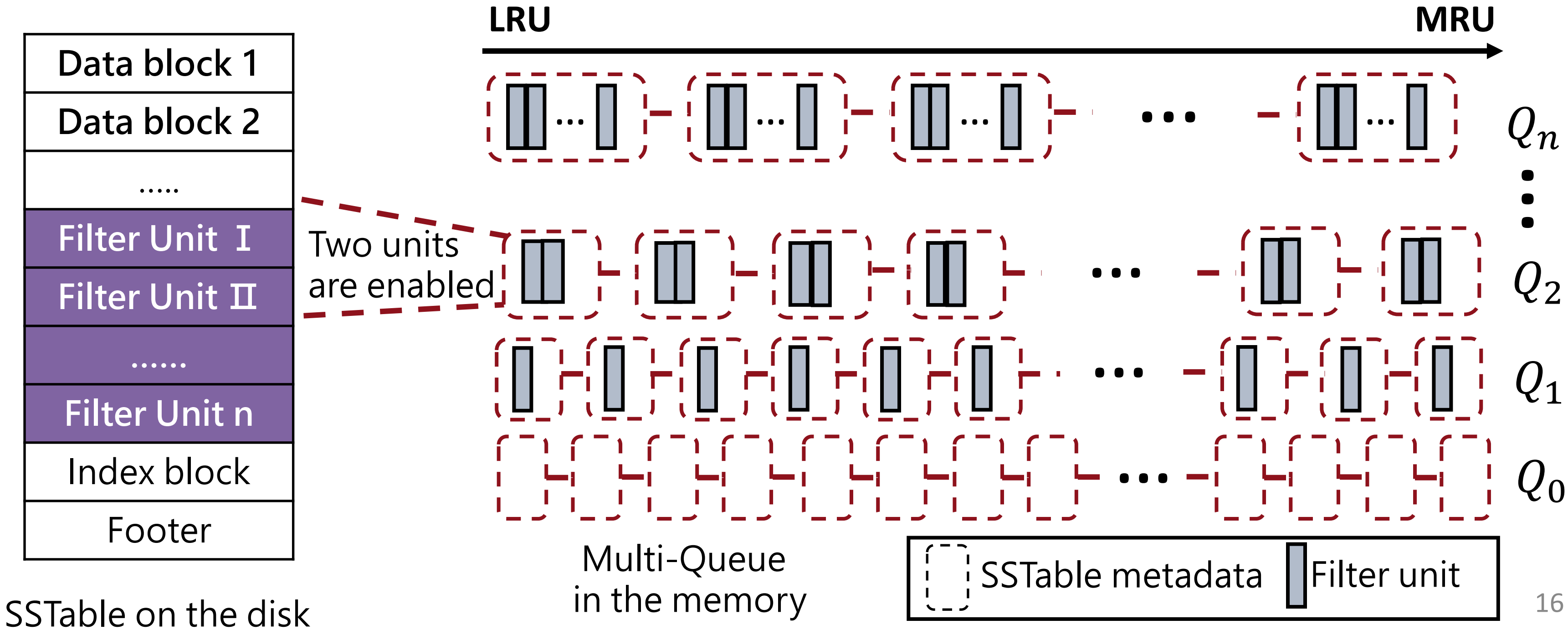
$$E[Extra\_IO] = \sum_{i=1}^{N} fp_i * f_i$$

  - Access frequency of SSTable i : f_i
  - False positive rate of the Bloom filter in SSTable i : fp_i
  - Number of SSTables in the KV store : N

ElasticBF estimates f_i in the runtime and adjusts fp_i accordingly so as to minimize $E[Extra\_IO]$

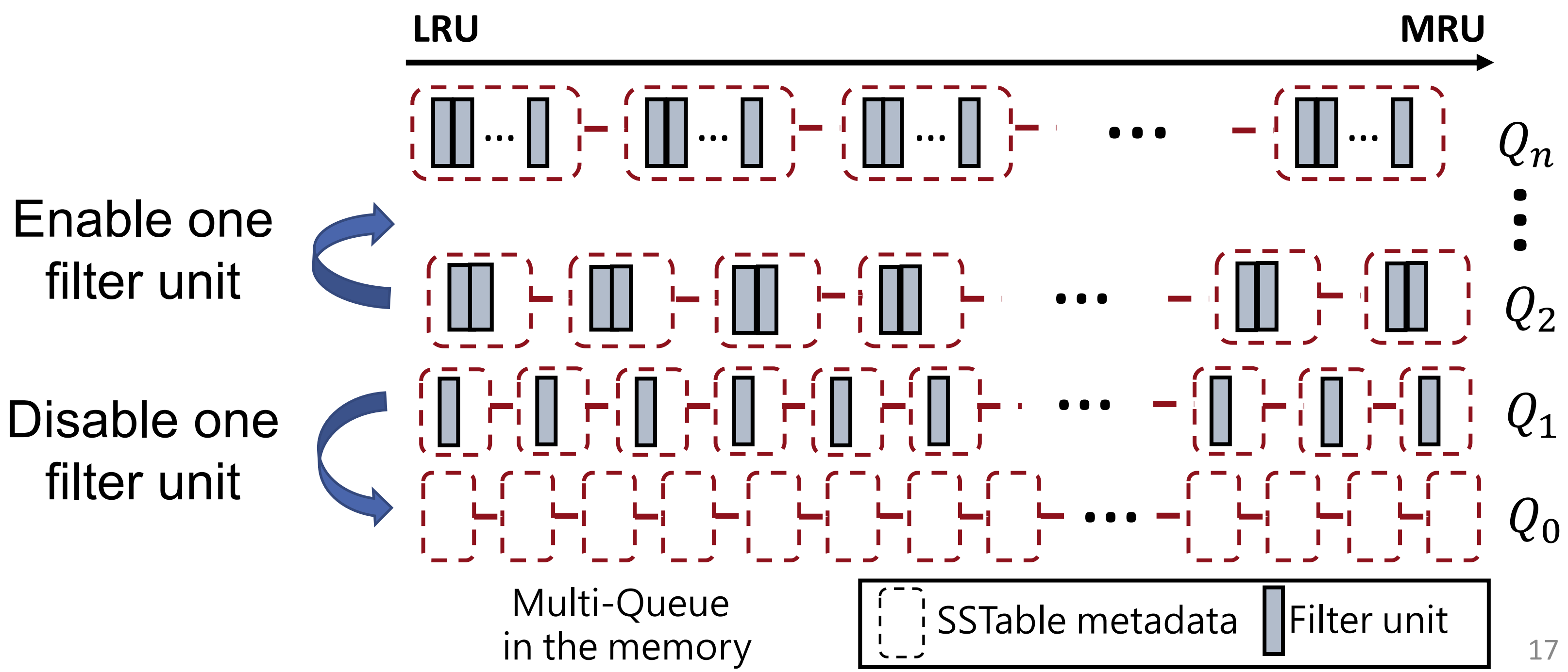
Multi-Queue

- ➢ Guides dynamic adjustment of the number of enabled filter units for each SSTable
  - ❑ Multiple least recently used queues (LRU)
  - ❑ Q_i corresponds to the SSTables with i filter units being enabled (Q_n : hottest SSTables, Q_0 : coldest SSTables)



Multi-Queue

- Dynamically adjust the filter units in Multi-Queue
  - Enable filter unit when the SSTable is accessed and $E[Extra\_IO]$ can be reduced
  - Disable filter unit according to expiring policy



Overhead Analysis

- Storage overhead

Size of KV pair	Size of SSTable	# KV pairs in a SSTable	bits-per-key	Space percent
1KB	2MB	2048	4	0.05%

- Computation overhead
  - Time of building filters : ~1%
  - Sufficient CPU resources
    - Multi-threading: generate multiple filter units simultaneously
- Memory overhead

Size of database	Number of SSTables	Memory overhead
100GB	50K	200KB

Experiment Setting

- Experiment environment
  - Machine

CPU	Disk	OS
Intel(R) Xeon(R) E5-2650 v4 @ 2.20GHz	Intel 3700 series SSD	CentOS 7.0/ Linux 3.10.0-5.14

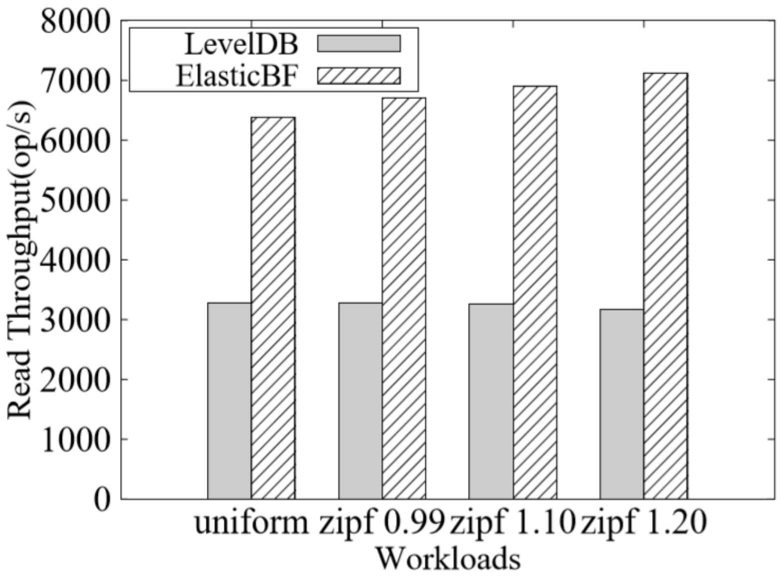
  - Workloads: YCSB

Size of KV pair	Size of database	Request Distribution	Zipfian skew	Zero lookup/ Non-zero lookup	Number of Get Requests
1024	100 GB	zipfian/uniform	0.99/1.1/1.2	1:1	1 million

Experiment Results

➢Read Performance

Throughput



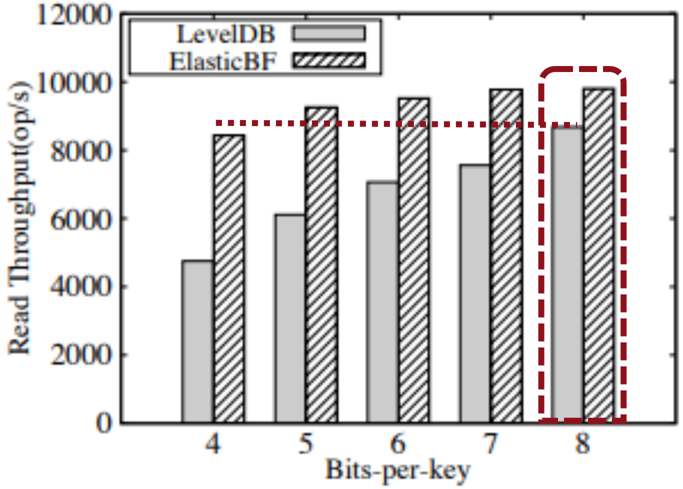
Number of I/Os for data access

	uniform	zipf 0.99	zipf 1.10	zipf 1.20
LevelDB	1525595	1585605	1634752	1667947
ElasticBF	628225	578553	550658	545345

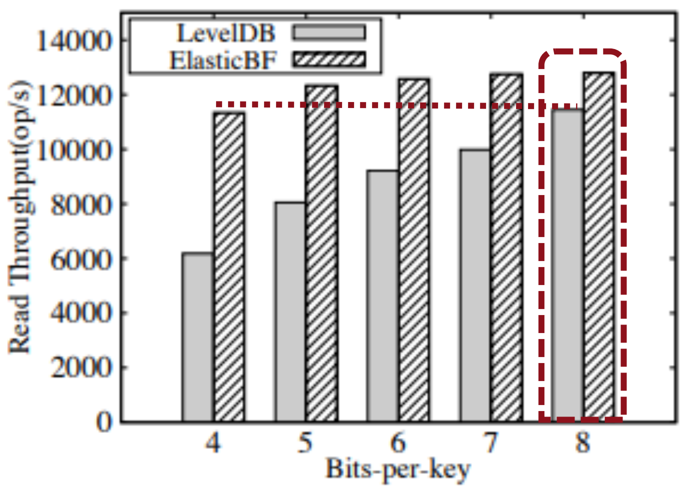
ElasticBF can achieve 1.94×-2.24× read throughput and greatly reduce the number of I/Os for data access compared to LevelDB

Experiment Results

➢Read Performance vs Memory Usage



(a) Uniform Workload

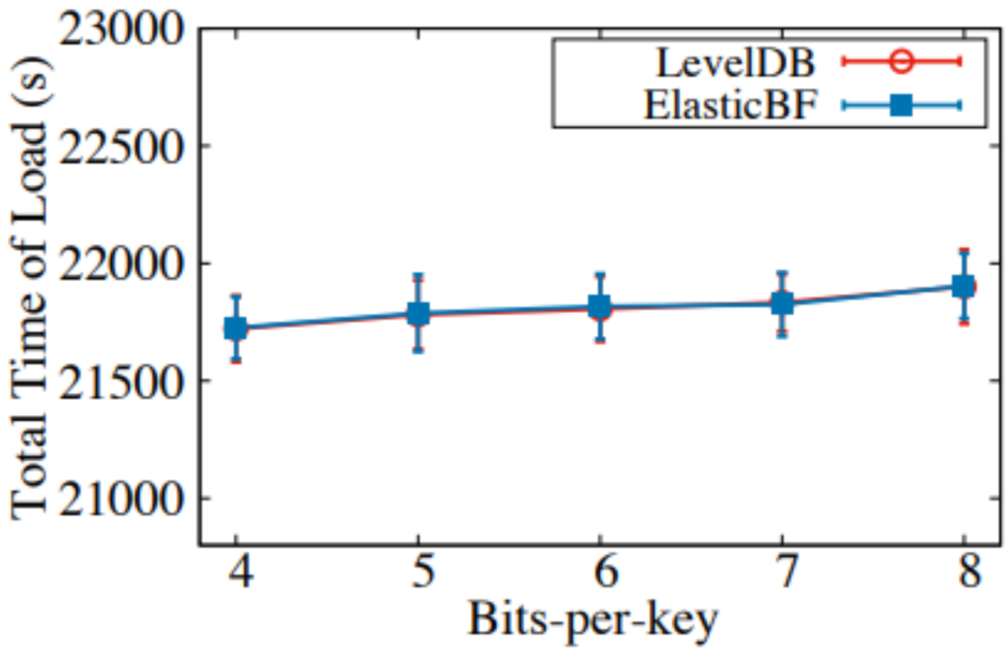


(b) Zipf0.99 Workload

ElasticBF can achieve a similar read performance with LevelDB with only a half memory usage

Experiment Results

- Write Performance
  - Load 100GB KV store



ElasticBF has almost the same write throughout with LevelDB

Conclusion

- LSM tree suffers from read amplification problem
  - Bloom filter reduces extra I/Os during read
  - Uniform Bloom filter design either suffers from high false positive rate or incurs large memory overhead

- We develop ElasticBF
  - An elastic scheme to dynamically adjust the Bloom filters in SSTables according to access frequency
  - Improves read performance with limited memory
  - Orthogonal to works optimizing LSM-tree structure

Thanks for your attention!

For any questions, please feel free to contact

Prof. Yongkun Li at USTC.

ykli@ustc.edu.cn

http://staff.ustc.edu.cn/~ykli/