ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores

Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, Yinlong Xu

University of Science and Technology of China

USENIX ATC 2019

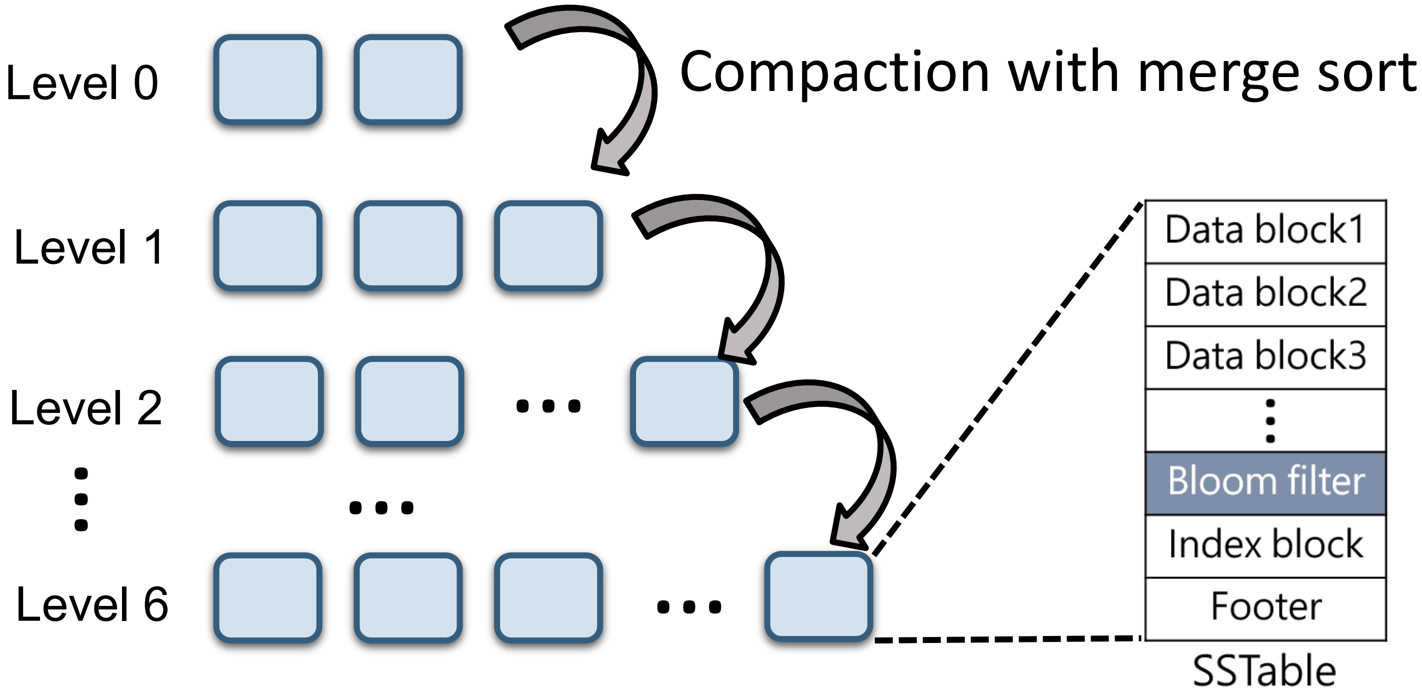
Background

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

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

LSM-tree-based KV Stores

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



Design Highlights

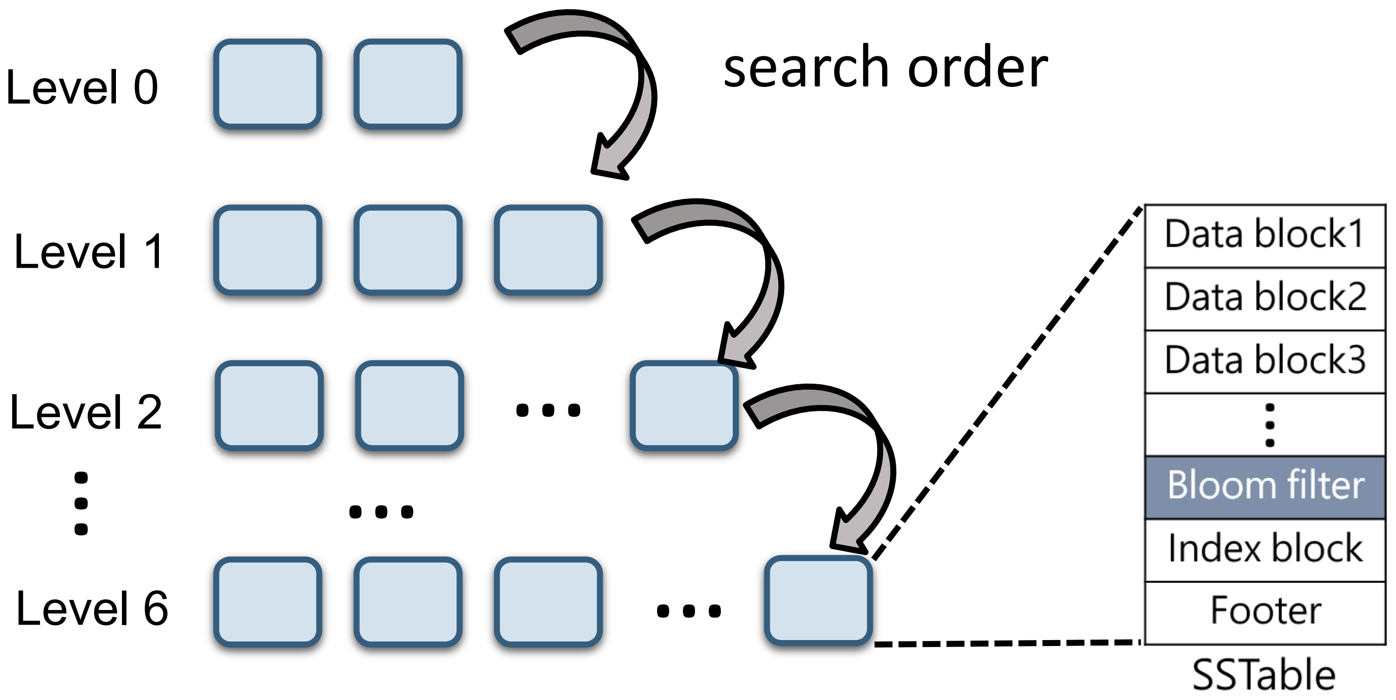
Layering

Log-structured writes

Sorted in each level

LSM-tree-based KV Stores

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



Read Amplification!!!

Key lookup: Check SSTables from lower levels to higher levels, one from each level (sorted)

Bloom Filters

improve read performance (also cached in memory)

Limitation of Bloom Filters

- Bloom filters suffer from false positive rate
  - False positive rate (FPR)： 0.6185^{b} (b: Bits-per-key)

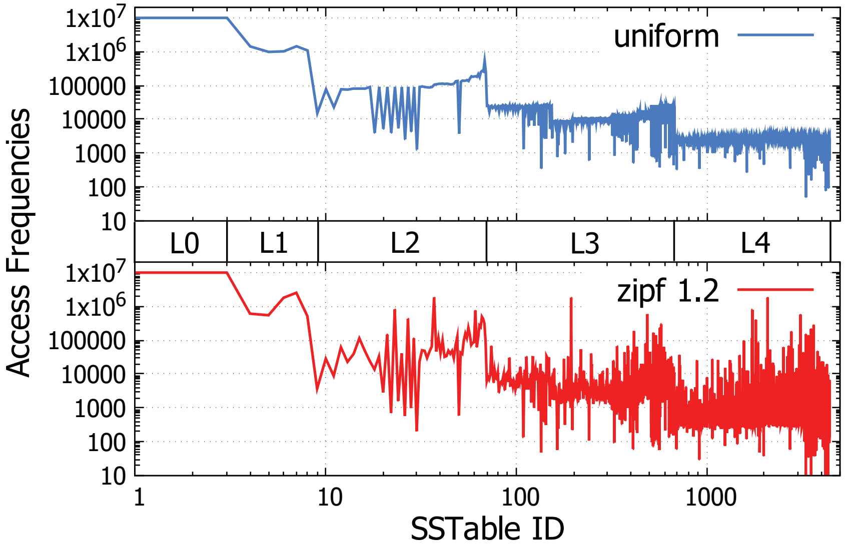
Bits-per-key	2bits	3bits	4bits	5bits	6bits
FPR	38.3%	23.7%	14.6%	9.1%	5.6%

- How to reduce false positive rate?
  - Allocate more bits for each key
  - Incur large memory overhead (as Bloom filters are cached in memory)

Question: how to improve the Bloom filter design with limited memory cost?

Main Idea

- **Observation**: unevenness of access frequencies
  - Vary from different levels, SSTables, or even different regions within an SSTable



Hot SSTables
More bits/key
Lower FPR

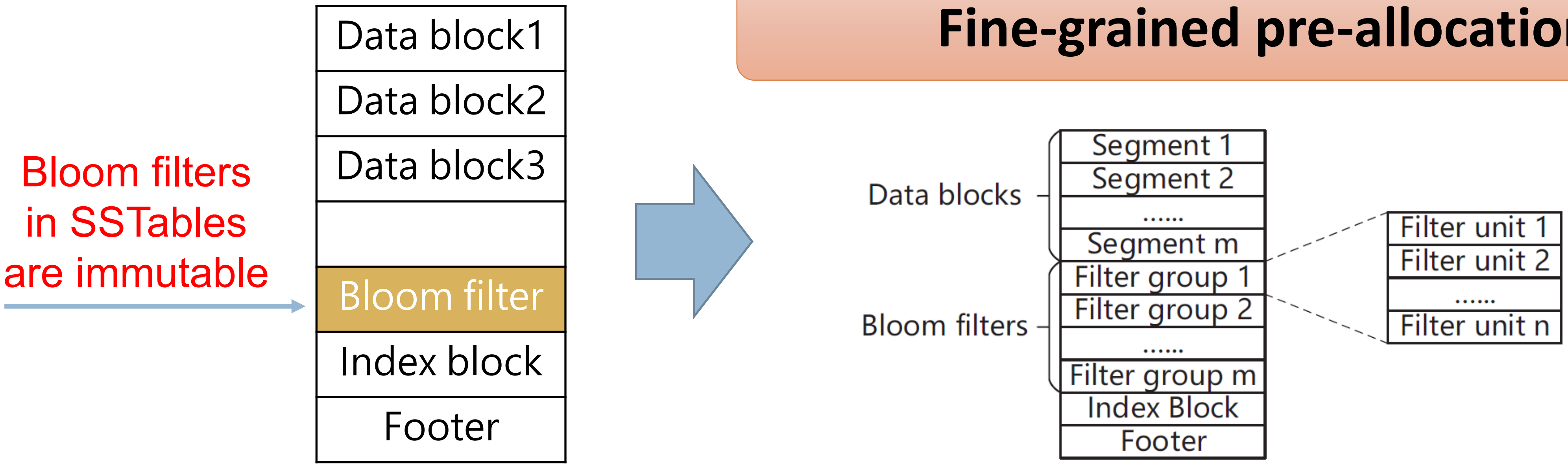
Cold SSTables
Fewer bits/key
Limited mem. usage

ElasticBF: Elastic Bloom filter management with locality awareness

ElasticBF Design

- **Challenge 1**: fixed data organization in SSTables limits BF adjustment

Fine-grained pre-allocation



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

ElasticBF Design

- **Challenge 2**: How to **determine** the most appropriate number of filter units for each SSTable and how to **realize** dynamical adjustment?

Determine: Cost-benefit analysis

Adjust Bloom filters only when the expected number of I/Os caused by false positive $E[Extra\_IO]$ can be reduced

$$E[Extra\_IO] = \sum_{\mathrm{i}=1}^{M} f_i * r_i$$

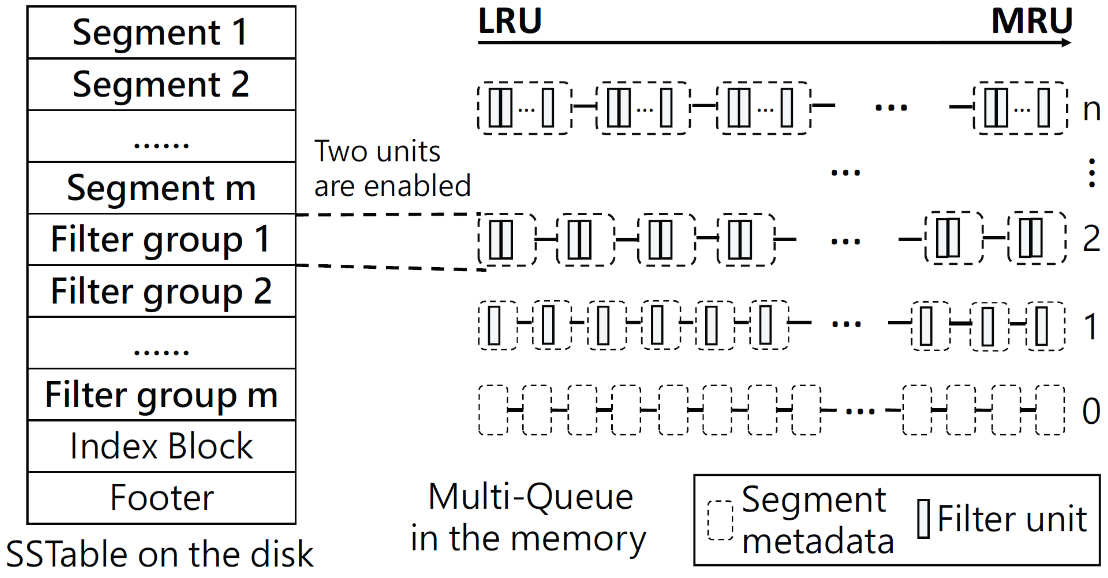
- M: # of segments in the system
- f_i: access frequency of segment i
- r_i: false positive rate of the BF allocated for seg. i

ElasticBF Design

➢**Challenge 2**: How to **determine** the most appropriate number of filter units for each SSTable and how to **realize** dynamical adjustment?

Adjust: in-memory multi-queue

Multiple LRU queues to realize dynamical adjustment



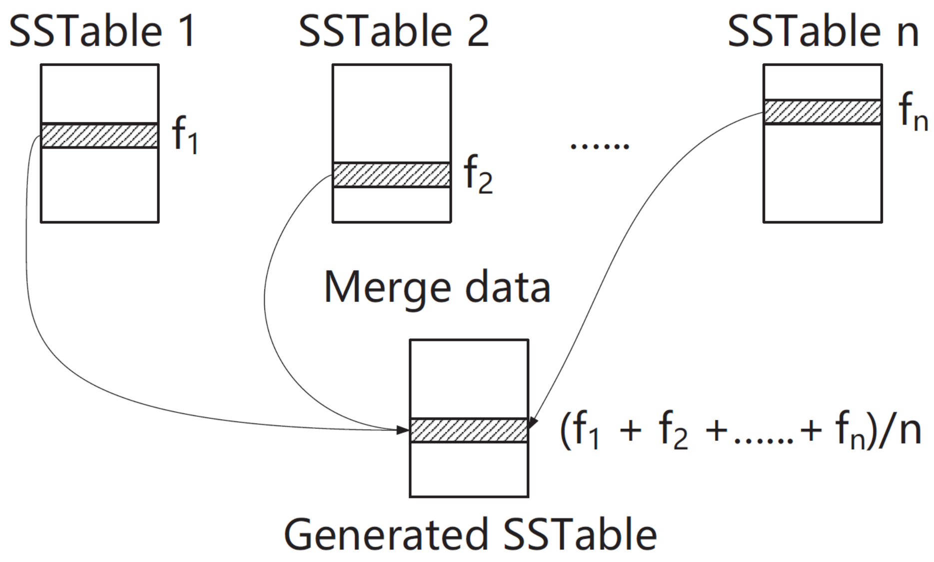
- **Upgrade:** each time when a segment is accessed, move to the MRU side
- **Downgrade:** search an "expired" segments from Q_n to Q_1 and move it to the next lower-level queue if $E[Extra\_IO]$ can be reduced by releasing one filter unit

ElasticBF Design

- ➢**Challenge 3**: Writes in mixed workloads may reset the hotness information (as compaction creates new SSTables)

Hotness inheritance

Estimate the hotness of new segments after compaction



① Find out involved old segments

② Estimate using the mean of the hotness of old segments

③ Enable an appropriate number of filter units based on the estimated hotness

Performance Evaluation

- ➢We implement ElasticBF in various KV stores: LevelDB/RocksDB/PebblesDB
- ➢Experiment setting
  - Machine

CPU	Mem/Disk	OS
Dell PowerEdge R730 12-cores Intel Xeon CPU E5-2650 v4 with 2.20GHz	64GB RAM 500GB SSD and 1TB 7200RPM HDD	Ubuntu 16.04 OS with Linux 4.15 kernel

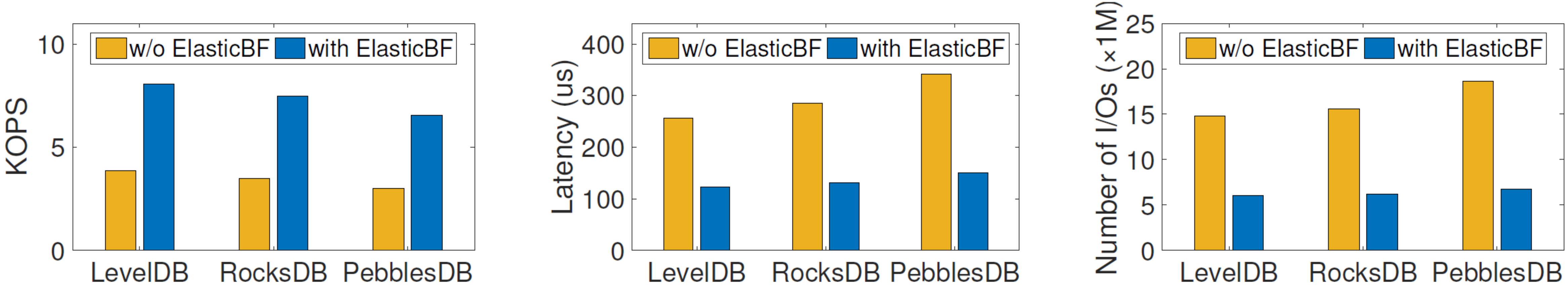
  - Micro-benchmarks: workloads generated by YCSB-C

Size of KV pair	Size of database	Request Distribution	Zipfian skew	Zero lookup/ Non-zero lookup	# of Get Req
1KB	100/400 GB	zipfian/uniform	0.99/1.1/1.2	1:1	10 million

  - YCSB benchmarks (six core workloads)

Micro-benchmarks

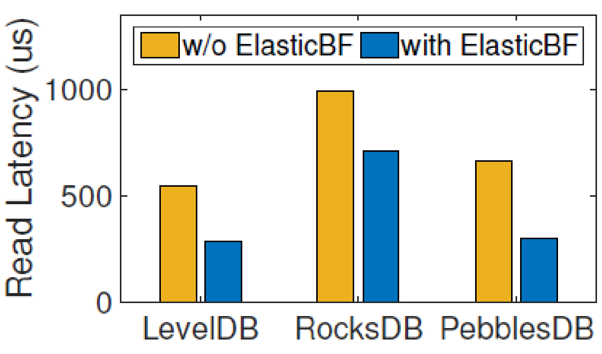
- How much improvement does ElasticBF achieve?
  - Compare read performance w/ and w/o ElasticBF (10M GET requests)



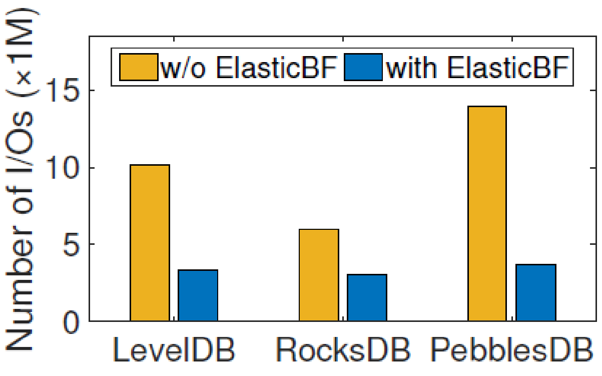
(a) Throughput (b) Latency (c) Total number of I/Os

ElasticBF increases the read throughput to >2x and reduces the latency by >50%, and also reduces the # of I/Os by ~60%

Micro-benchmarks

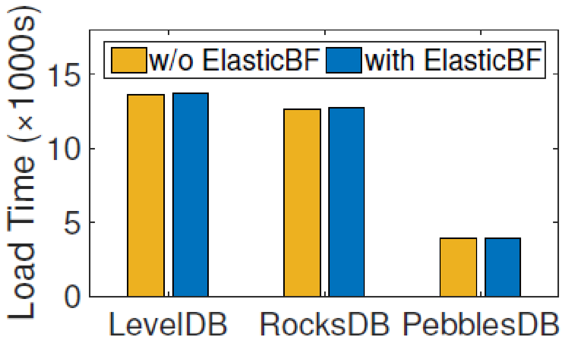


(a) Read latency (50% reads)

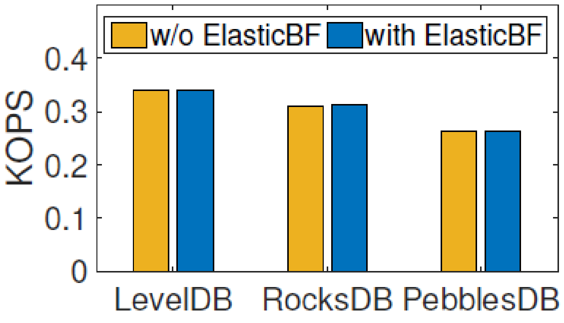


(b) Number of I/Os (50% reads)

- ➢Mixed workload
  - Still remarkable improvement



(a) Time to load the KV store

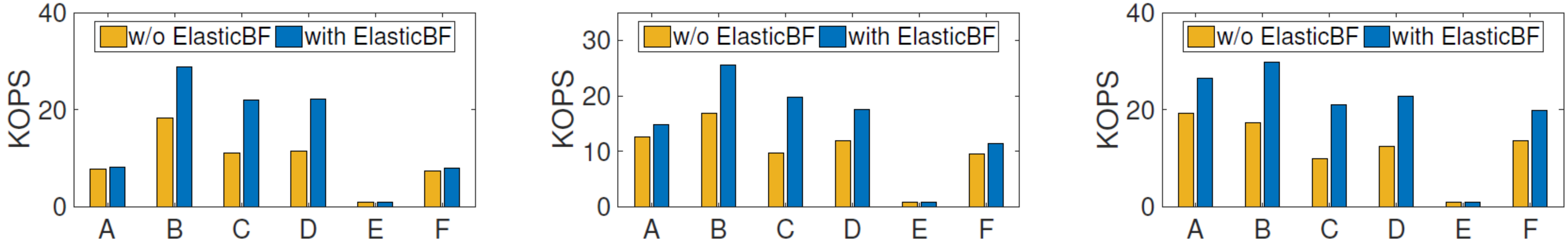


(b) Throughput of range query

- ➢PUT and SCAN performance
  - Negligible impact

YCSB Benchmarks

- ➢YCSB benchmarks

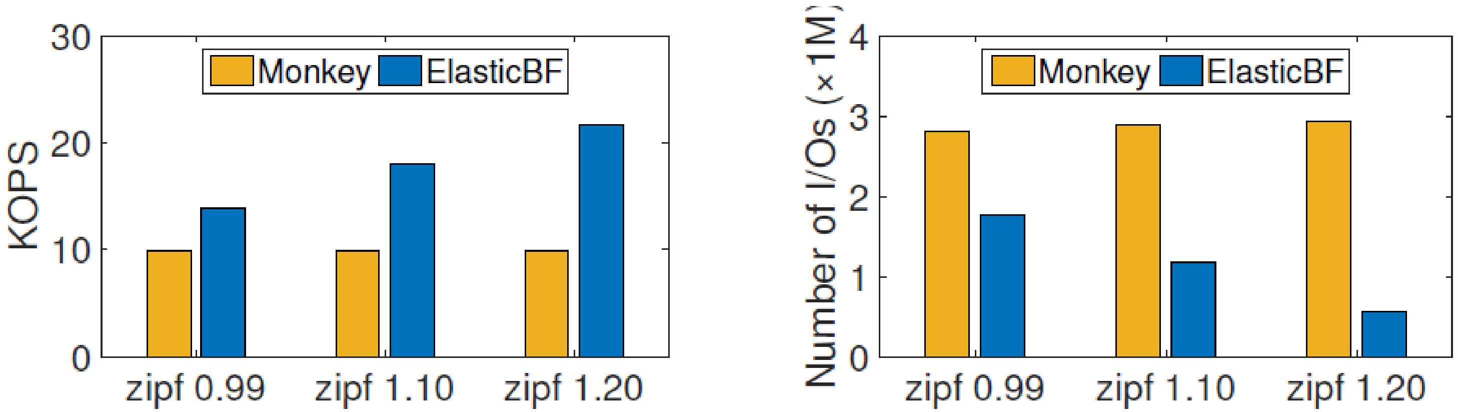


(a) LevelDB (b) RocksDB (c) PebblesDB

ElasticBF improves read throughput under read-dominant workloads (B: 95% read, C: 100% read, D: 95% read)

Comparison with Monkey

- ➢Monkey: coarse-grained scheme (even BF allocation in each level) w/o dynamical adjustment



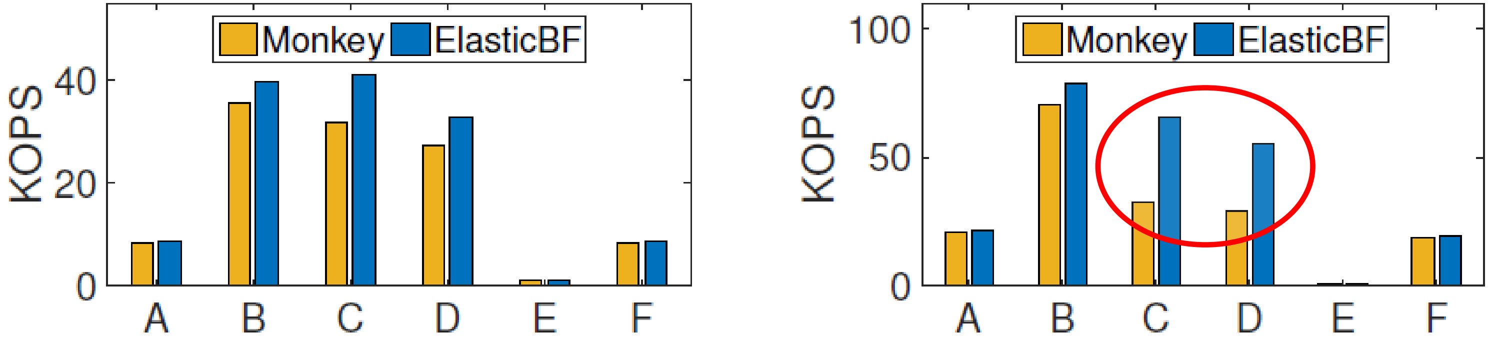
(a) Thpt. under 100GB database (b) Total number of I/Os

Micro-benchmark: 10M GET to 100GB KV store

ElasticBF further increases the throughput to 1.39× – 2.20×

Comparison with Monkey

- ➢Monkey: coarse-grained scheme (even BF allocation in each level) w/o dynamical adjustment



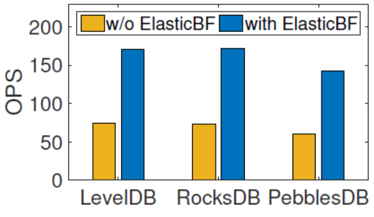
(a) Throughput with Zipf 0.99 (b) Throughput with Zipf 1.2

YCSB benchmark: 10M GET to 100GB KV store

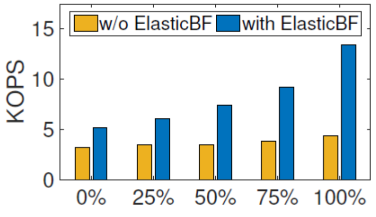
ElasticBF further increases the throughput up to ~2× under read-dominant workloads with high skewness

Impact of System Configurations

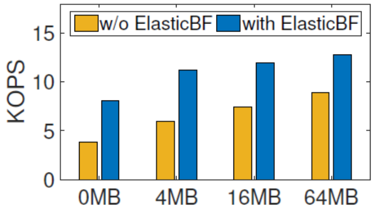
- ➢Impact of
  - Hard disk
  - Zero lookup ratio
  - Block cache size
  - KV pair size
  - Database size
  - Segment size
  - Filter unit size



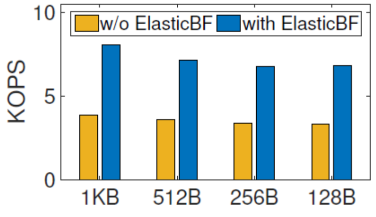
(a) Performance under HDD



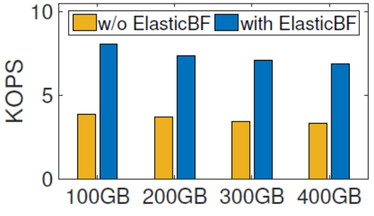
(b) Impact of zero lookup ratio



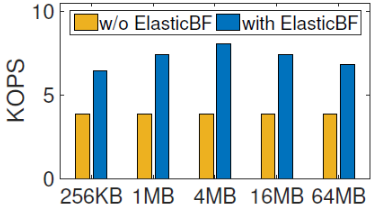
(c) Impact of block cache size



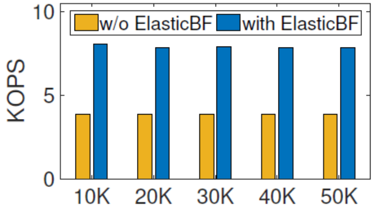
(d) Impact of KV pair size



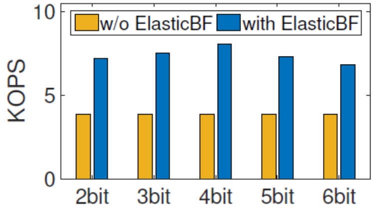
(e) Impact of database size



(f) Impact of segment size



(g) Impact of `lifeTime`



(h) Impact of filter unit size

➢Please refer to our paper for detailed results

Conclusion

- LSM-tree based KV stores suffer from read amplification problem
  - Bloom filters reduce extra I/Os and improve read performance
  - 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 Bloom filters, so it improves read performance with limited memory
  - Orthogonal to the optimizations of the LSM-tree structure, so it can be deployed in various existing KV stores

Thanks for your attention!

For any questions, please feel free to contact

Prof. Yongkun Li@USTC

ykli@ustc.edu.cn

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