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
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
- ...

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

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

Compaction incurs write amplification.

Fully sorted in each level.
The most common design of KV stores is based on LSM-tree (log structured merge tree).

Looking up a key requires multiple I/O requests as it may require to search in multiple levels (read amplification).
One typical implementation of LSM tree

- Focus on data layout on disk

Level 0 (8MB)
Level 1 (10MB)
Level 2 (100MB)
…
Level 6 (1TB)

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)

<table>
<thead>
<tr>
<th>Bits-per-key</th>
<th>2bits</th>
<th>3bits</th>
<th>4bits</th>
<th>5bits</th>
<th>6bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR</td>
<td>40%</td>
<td>23.7%</td>
<td>14.7%</td>
<td>9.2%</td>
<td>5.6%</td>
</tr>
</tbody>
</table>
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

```
<table>
<thead>
<tr>
<th>Data block1</th>
<th>Data block2</th>
<th>Data block3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bloom filter</td>
</tr>
<tr>
<td></td>
<td>Index block</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Footer</td>
<td></td>
</tr>
<tr>
<td>SSTable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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: \( \prod_{i=1}^{n} 0.6185^{b_i} = 0.6185^b \) (\( \sum_{i=1}^{n} b_i = b \))
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

Data organization in SSTable does not change

Elastic feature: false positive rate can be dynamically adjusted
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

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

- 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 \( E[\text{Extra\_IO}] \)
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

![Diagram showing LRU and MRU queues with filter units enable and disable options.](image)
Overhead Analysis

Storage overhead

<table>
<thead>
<tr>
<th>Size of KV pair</th>
<th>Size of SSTable</th>
<th># KV pairs in a SSTable</th>
<th>bits-per-key</th>
<th>Space percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>2MB</td>
<td>2048</td>
<td>4</td>
<td>0.05%</td>
</tr>
</tbody>
</table>

Computation overhead

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

Memory overhead

<table>
<thead>
<tr>
<th>Size of database</th>
<th>Number of SSTables</th>
<th>Memory overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>100GB</td>
<td>50K</td>
<td>200KB</td>
</tr>
</tbody>
</table>
Experiment Setting

- **Experiment environment**
  - **Machine**

<table>
<thead>
<tr>
<th>CPU</th>
<th>Disk</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel(R) Xeon(R) E5-2650 v4 @ 2.20GHz</td>
<td>Intel 3700 series SSD</td>
<td>CentOS 7.0/ Linux 3.10.0-5.14</td>
</tr>
</tbody>
</table>

- **Workloads: YCSB**

<table>
<thead>
<tr>
<th>Size of KV pair</th>
<th>Size of database</th>
<th>Request Distribution</th>
<th>Zipfian skew</th>
<th>Zero lookup/ Non-zero lookup</th>
<th>Number of Get Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>100 GB</td>
<td>zipfian/uniform</td>
<td>0.99/1.1/1.2</td>
<td>1:1</td>
<td>1 million</td>
</tr>
</tbody>
</table>
ElasticBF can achieve $1.94 \times - 2.24 \times$ read throughput and greatly reduce the number of I/Os for data access compared to LevelDB.
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/