SuRF: Implementation and RocksDB

This paper is a report of implementation of original paper: SuRF: Practical Range Query Filtering with Fast Succinct Tries

Dhruvil Gandhi
Seidenberg School of CSIS
Pace University
New York, NY
Email: dgandhi@pace.edu

Gunjan Asrani
Seidenberg School of CSIS
Pace University
New York, NY
Email: gasrani@pace.edu

***Abstract*—We present implementation of Succinct Range Filter (SuRF), a fast and compact data structure for approximate membership tests. It supports single-key lookups and common range queries: open-range queries, closed-range queries, and range counts based on a new data structure called the Fast Succinct Trie (FST) that matches the point and range query performance of state-of-the-art order-preserving indexes, while consuming only 10 bits per trie node. The false positive rates in SuRF can be modified to satisfy different application needs. We are evaluating SuRF in RocksDB as a replacement for its Bloom filters to reduce I/O by filtering requests before they access on-disk data structures; the data is 100GB. The experiment uses YCSB data.**

***Index Terms*—Filters, ARF, Bloom Filter, SuRF, rocksdb, FST**

I. Background

There are different approximate query membership data structures designed and evaluated. Filters answers approximate membership of a query. Filters act as a guard when accessing data from devices which have high latency and limited bandwidth. Traditionally designed filters only support point queries, complete keys must be provided to the filter to test for membership. The paper [1] proposes a new filter, SuRF or Succinct Range Filter which filters with both point queries (complete keys) and range queries(range of keys or partial keys). SuRF uses suffixed keys to improve false positive rate of the filter. The paper [1] has experimentation implemented and evaluated for SuRF implemented in C++ [3] and also tests performance [4] of RocksDB [2] by Facebook for point queries, range queries on single threaded and multi-threaded implementation. They evaluation states 1.5x to 5x improvement in RocksDB using SuRF over RocksDB with Bloom Filter.

II. Motivation

SuRF is memory efficient [1], improves throughput and reduces I/O substantially. It seems promising technique to optimize database. To implement and re-evaluate the proposed solution, we tried to replicated and reproduce the results. RocksDB by facebook [2] uses Bloom Filter, a point query filter. We try to replicate the experiment, replacing Bloom Filter with SuRF and compare them.

III. Design

The experiment code is available on github for SuRF [3] and RocksDB [2]. We first try to evaluate both.

The basic version of SuRF (SuRF-Base) stores the minimum length key prefixes such that it can uniquely identify each key. Specifically, SuRF-Base only stores an additional byte for each key beyond the shared prefixes.

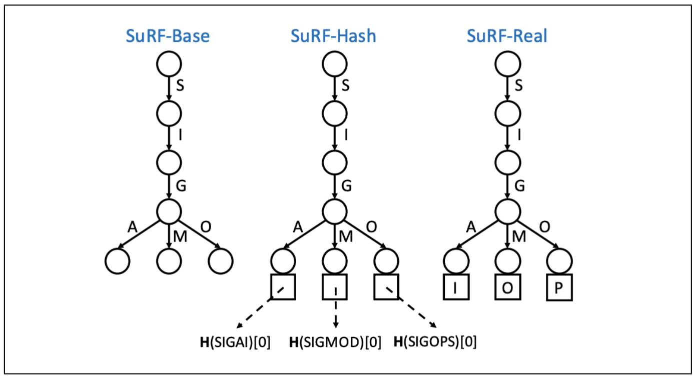

Fig. 1: Different SuRF versions

Different variations of SuRF are as

- Basic SuRF
- SuRF with Hashed Key Suffixes
- SuRF with Real Key Suffixes
- SuRF with Mixed Key Suffixes

IV. Implementation

We carried out a set of experimentation for both SuRF and the example application for RocksDB. First we implemented SuRF basic code to compare Bloom Filter, No Filter and different variations of SuRF. Then we implemented different variants of SuRF with multi-threading support. Data from Yahoo's Cloud Serving Benchmark[6] is used. The experiments is not implemented for email list.

Our first run used the following configuration(t3.large on AWS [5]):

- 2.5 GHz Intel Scalable Processor
- Intel AVX†, Intel AVX2†, Intel Turbo
- EBS Optimized(SSD for 300 I/Ops)
- Enhanced Networking†
- 2 vCPUs
- 8GB of memory

This configuration ran out of memory on first run.

We switched to DigitalOcean with following configuration:

- Intel(R) Xeon(R) CPU E5-2650L v3 1.80GHz
- 6vCPUs
- 16GB Multi-EEC
- 160GB SSD
- 8GB of memory

A. RocksDB Experiment

After successful run of the SuRF code, we implemented the experiment carried out with RocksDB. The experiment implements SuRF in RocksDB and uses 100GB of data. Different queries are carried out to test data with no-filter, bloom filter and SuRF. Throughput and seek timings in different scenarios recorded. The data is tested with 100% positive and 50% empty queries.

V. Evaluation

A. SuRF Single Threaded

When running single threaded operations, the output was seen as:

TABLE I
POINT QUERY RESULT

Operation	Throughput	FPR
Bloom Filter	4.18142	0.00771089
SuRF	2.25069	0.0414119
SuRFHash	1.9057	0.00162597
SuRFReal	1.95216	0.0054646
SuRFMixed	1.61184	0.00951337

B. SuRF Multi-Threaded

When running different filters multi-threaded, we get the following throughput graph:

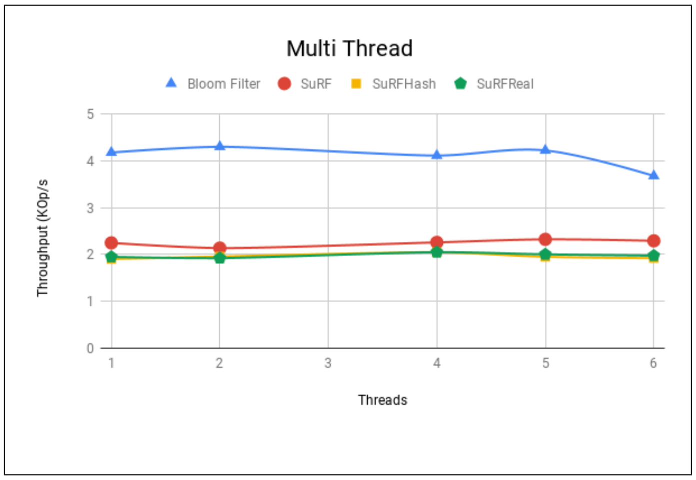

Fig. 2: Multi Threaded

C. RocksDB

When running with RocksDB, we first run it with point queries for No Filter, Bloom FIiter, and SuRF. We observe that Bloom Filter has better throughput with Bloom Filter. This can be due to cached data. This needs to be examined again.

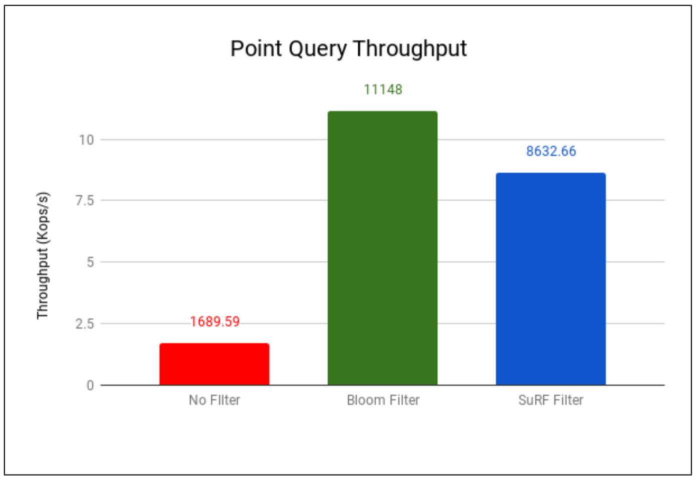

Fig. 3: Point Query RocksDB

We perform next set of evaluations with 50% success range queries for No Filter, Bloom FIiter, and SuRF. We clear system cache before running each experiment.

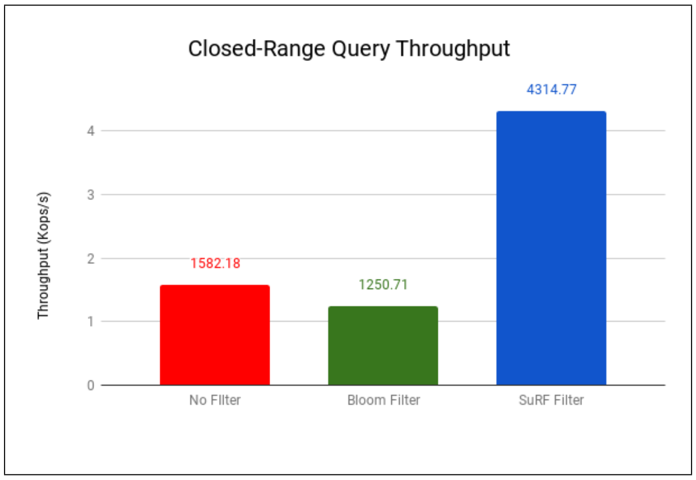

Fig. 4: 50% empty results query RocksDB

We observe that SuRF significantly outperforms Bloom Filter and without filter operations. The I/O could not be obtained on the cloud due to logical disk structure naming.

D. Limitation

The only limitation in implementation of this is there is lack of resources on the code developed.

E. Comparision and Conclusion

After implementing all the experiments, we conclude that SuRF is a promising filter for databases with improvements in throughput. In case of RocksDB, the I/O was monitored with monitoring metrics that DigitalOcean provides. The number of operations were reduced when using SuRF. We were unable to compare it to ARQ as the build step was not executed.

VI. Future Work

We will like to implement more data-points and test for SuRF and compare it with ARF. We will like to run more tests with RocksDB.

Acknowledgment

We would like to thanks Dr. Juan Shan at Pace University. And DigitalOcean for final minutes changes in the billing plan to allow acclerated execution. The git repository for this work is available at [7]

References

[1] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018.*SuRF: Practical Range Query Filtering with Fast Succinct Tries*. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). ACM, New York, NY, USA, 323-336. DOI: https://doi.org/10.1145/3183713.3196931
[2] 2018. RocksDB SuRF Implementation Github https://github.com/efficient/rocksdb
[3] 2018. SuRF Repository https://github.com/efficient/SuRF
[4] 2018. RocksDB Repository by Facebook https://github.com/facebook/rocksdb
[5] 2015. AWS EC2 intance https://aws.amazon.com/ec2/instance-types/t3/
[6] 2018. Yahoo Cloud Serving Benchmark https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/?guccounter=1
[7] 2018. SuRF-Report Github https://github.com/dhruv857/rocksdb
[8] 2018. Working rocksdb Repo https://github.com/dhruv857/rocksdb