SuRF: PRACTICAL RANGE FILTERING WITH FAST SUCCINCT TRIES

Huanchen Zhang
Hyeontaek Lim, Viktor Leis, David G. Andersen
Michael Kaminsky, Kimberly Keeton, Andrew Pavlo
Filters answer approximate membership queries
Filters answer approximate membership queries
Filters answer approximate membership queries
Filters answer approximate membership queries

Billionaire?

YES, 100%

No False Negatives
Filters answer approximate membership queries
Filters answer approximate membership queries

NO, 99%

Billionaire ?
Filters answer approximate membership queries

Billionaire?

NO, 99%

YES, 1%
Filters answer approximate membership queries

Billionaire

NO, 99%

YES, 1%

?
Filters answer approximate membership queries

NO, 99%

Billionaire ?

YES, 1%

False Positive Rate
Filters pre-reject most negative queries
Filters pre-reject most negative queries

Queries

NO

Probably YES

Local Memory

Slow Devices
Existing filters only support point filtering

**Point Filtering**

```sql
SELECT * FROM Billionaires
WHERE LastName = 'Pavlo'
```

- **Bloom Filter** *(1970)*
- **Quotient Filter** *(2012)*
- **Cuckoo Filter** *(2014)*
Existing filters only support point filtering

**Point Filtering**

<table>
<thead>
<tr>
<th>Filter</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter</td>
<td>1970</td>
</tr>
<tr>
<td>Quotient Filter</td>
<td>2012</td>
</tr>
<tr>
<td>Cuckoo Filter</td>
<td>2014</td>
</tr>
</tbody>
</table>

**Range Filtering**

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM Billionaires WHERE LastName = 'Pavlo'</td>
</tr>
<tr>
<td>SELECT * FROM Billionaires WHERE LastName LIKE 'Pav%'</td>
</tr>
</tbody>
</table>
Existing filters only support point filtering

<table>
<thead>
<tr>
<th>Point Filtering</th>
<th>Range Filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM Billionaires WHERE LastName = 'Pavlo'</td>
<td>SELECT * FROM Billionaires WHERE LastName LIKE 'Pav%'</td>
</tr>
<tr>
<td>Bloom Filter (1970)</td>
<td></td>
</tr>
<tr>
<td>Quotient Filter (2012)</td>
<td></td>
</tr>
<tr>
<td>Cuckoo Filter (2014)</td>
<td></td>
</tr>
</tbody>
</table>
Our solution: **Succinct Range Filters (SuRF)**

First practical, general-purpose range filter

**SMALL:** close to theoretic minimum

- 64-bit integer keys, 1% false positive rate: $\approx 12$ bits per key

**FAST:** comparable to fastest trees

- 10 million 64-bit integer keys: $\approx 200$ ns per query

**USEFUL:** evaluated in RocksDB

- speed up range queries by up to $5x$
Starting point: a complete trie
Starting point: a complete trie
Make it smaller: a truncated trie
Make it smaller: a truncated trie
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits
Use suffix bits to reduce false positive rate

**Hashed Suffix Bits**

Hashed Suffix Bits: SIGMETRICS

**Real Suffix Bits**

Real Suffix Bits: SIGMETRICS

0xC8 0x20 0x06

0xC8 0x20 0x06
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits

SIGMETRICS

K

G

O

0xC8

0x20

0x06

0x18

SIGMETRICS

K

G

O

D

0x18

E
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits

0xC8 0x20 0x06

D O P
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits

Each bit reduces FPR by half
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

```
+---+---+---
| S | I | G |
+---+---+---
  | K |
+---+---+---
  | M | O |
+---+---+---
0xC8 0x20 0x06
```

Each bit reduces FPR by half

Real Suffix Bits

```
+---+---+---
| S | I | G |
+---+---+---
  | K |
+---+---+---
  | M | O |
+---+---+---
D O P
```

Cannot help range queries
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

Real Suffix Bits

Each bit reduces FPR by half

Benefit point & range queries

Cannot help range queries
Use suffix bits to reduce false positive rate

Hashed Suffix Bits

- Each bit reduces FPR by half
- Cannot help range queries

0xC8 0x20 0x06

Real Suffix Bits

- Benefit point & range queries
- Weaker distinguishability
Succinct Data Structure

... uses an amount of space that is “close” to the information-theoretic lower bound, but still allows efficient query operations. [wikipedia]
SuRF’s encoding is small and fast

1. **Small**
   - \( \approx 10 + \text{suffix} \) bits per key for 64-bit integers
   - \( \approx 14 + \text{suffix} \) bits per key for emails

2. **Fast**
   - Matches state-of-the-art pointer-based trees
Bloom filters speed up point queries in RocksDB

Cached Filters
B, B, B, ...

L_{N-2} ...
L_{N-1} ...
L_N ...

..., 6, 20, ...
..., 12, 21, ...
..., 11, 19, ...

→ SSTable
Bloom filters speed up point queries in RocksDB

Cached Filters

\[ L_{N-2} \ldots \]
\[ L_{N-1} \ldots \]
\[ L_N \ldots \]

\[ \ldots, 6, 20, \ldots \]
\[ \ldots, 12, 21, \ldots \]
\[ \ldots, 11, 19, \ldots \]

\( \text{GET}(16) \)
Bloom filters speed up point queries in RocksDB

Cached Filters
B, B, B, ...

NO
GET(16)

\( L_{N-2} \)
\( \ldots \)
\( \ldots, 6, 20, \ldots \)
\( B \)

\( L_{N-1} \)
\( \ldots \)
\( \ldots, 12, 21, \ldots \)
\( B \)

\( L_N \)
\( \ldots \)
\( \ldots, 11, 19, \ldots \)
\( B \)
Bloom filters can’t help range queries in RocksDB

SEEK(14, 18)
Bloom filters can't help range queries in RocksDB

Cached Filters
B, B, B, ...

SEEK(14, 18)

..., 6, 20, ...
B

..., 12, 21, ...
B

..., 11, 19, ...
B
SuRFs can benefit both point and range queries

Cached Filters
S, S, S, ...

\[ L_{N-2} \] \[\ldots\] \[\ldots, 6, 20, \ldots\] S

\[ L_{N-1} \] \[\ldots\] \[\ldots, 12, 21, \ldots\] S

\[ L_{N} \] \[\ldots\] \[\ldots, 11, 19, \ldots\] S
SuRFs can benefit both point and range queries

Cached Filters
S, S, S, ...

NO
GET(16)
SEEK(14, 18)
Evaluation setup: a time-series benchmark

Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload
Evaluation setup: a time-series benchmark

Queries:

GET(t)

SEEK(t₁, t₂)

Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload
Evaluation setup: a time-series benchmark

Queries:
- GET(t)
- SEEK(t₁, t₂)

Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload

System Config

Dataset: ≈100 GB on SSD
DRAM: 32 GB
Evaluation setup: a time-series benchmark

Queries:
- GET(t)
- SEEK(t₁, t₂)

Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload

System Config
- Dataset: ≈100 GB on SSD
- DRAM: 32 GB

Filter Config
- Bloom filter: 14 bits per key
- SuRF: 4-bit real suffix
SuRFs still benefit point queries in RocksDB

All-false point queries

Throughput (Kops/s)

- No Filter
- Bloom Filter
- SuRF

Worst-case Gap
SuRFs speed up range queries in RocksDB

Throughput (Kops/s)

Percent of queries with empty results

SuRF

No Filter/Bloom Filter
SuRFs speed up range queries in RocksDB

Throughput (Kops/s)

Percent of queries with empty results

SuRF

5x

No Filter/Bloom Filter
Conclusion

SuRF is a fast and compact data structure optimized for range filtering

github.com/efficient/SuRF

[Demo] rangefilter.io
Backup Slides
## Comparing ARF to SuRF

Experiment: insert 5M 64-bit integers, 10M Zipf-distributed range queries (ARF uses 2M queries for training)

<table>
<thead>
<tr>
<th></th>
<th>ARF</th>
<th>SuRF</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits per Key (held constant)</td>
<td>14</td>
<td>14</td>
<td>-</td>
</tr>
<tr>
<td>Range Query Throughput (Mops/s)</td>
<td>0.16</td>
<td>3.3</td>
<td>20x</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>25.7</td>
<td>2.2</td>
<td>12x</td>
</tr>
<tr>
<td>Build Time (s)</td>
<td>118</td>
<td>1.2</td>
<td>98x</td>
</tr>
<tr>
<td>Build Memory (GB)</td>
<td>26</td>
<td>0.02</td>
<td>1300x</td>
</tr>
<tr>
<td>Training Time (s)</td>
<td>117</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Training Throughput (Mops/s)</td>
<td>0.02</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
LOUDS-Sparse encoding example

LOUDS-Sparse

Label: a i d h t f t
Has-child: 1 1 0 0 0 0 0 0
Structure: 1 0 1 0 0 1 0
Value: v1 v2 v3 v4 v5

Theoretic Limit \( \approx 9.4N \) bits

10N bits

\[
moveToChild(p) = select(S, \text{rank}(HC, p) + 1)
\]
LOUDS-DS trades small space for performance

Hot

LOUDS-Dense

LOUDS-Sparse

< 1% space overhead

3x speed-up