Grafite: Taming Adversarial Queries with Optimal Range Filters

Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra
Range filters

Given a set $S$ of $n$ keys, a range filter is a space-efficient data structure that answers range emptiness queries

$[a, b] \cap S = \emptyset? \quad [a', b'] \cap S = \emptyset? \quad [a'', b''] \cap S = \emptyset?$

Not empty \hspace{1cm} Empty \hspace{1cm} Not empty

...with a false positive probability of at most $\varepsilon$

• Generalise Bloom filters from point to range queries
• Reduce I/Os of range queries in LSM-based storage engines
State-of-the-art range filters

ARF [VLDB ’13], SuRF [SIGMOD ’18], Rosetta [SIGMOD ’20], Proteus [SIGMOD ’22], SNARF [VLDB ’22], bloomRF [EDBT ’23], REncoder [ICDE ’23], Oasis [VLDB ’24], GRF [SIGMOD ’24]

Highly complex
Sophisticated designs, hard to evaluate and deploy

Fragile
Inconsistent FPR and query times across different datasets and workloads

Goswami et al. [SODA ’15] gave theoretically optimal upper and lower bounds
Correlated queries

Most range filters suffer from high false positive rates when query ranges are close to the input keys

The most expected queries are also adversarial ones!!!
Correlated queries

![Graph showing correlation degree vs false positive rate and time per query for different tools: SuRF, SNARF, REncoder, Rosetta, Proteus, Grafite.](image)

- False Positive Rate vs Correlation Degree
- Time [ns/query] vs Correlation Degree
Grafite: hash

1. Apply the locality-preserving hash function by Goswami et al. [SODA ’15] to the input keys

\[ h(x) = (x + q([x/r])) \mod r \]

Properly chosen to bound FPR and space
Grafite: hash + compress

2. Compress the hash codes with the Elias-Fano integer code

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \ 0 & 0 & 0 & 0 & 1 & 1 \ 0 & 0 & 1 & 1 & 0 & 1 \ 0 & 1 & 0 & 1 & 0 & 1 \ 1 & 0 & 0 & 0 & 0 & 1 \ 1 & 1 & & & & \\
\end{array}
\]

[0, r)

Store the low parts explicitly
Encode the high parts with a bitvector
Grafite: hash + compress + query

3. Solve queries in hash space

\[ [a, b] \cap S = \emptyset? \]

\[ [h(a), h(b)] \]

\[ [0, r) \]

\[ [0, u) \]
Grafite: hash + compress + query

3. Solve queries in hash space

\[[a, b] \cap S = \emptyset?\]

\[[h(a), h(b)]\]

\[[0, r)\]

Not empty
Grafite's theoretical guarantees

**Theorem.** Given a space budget of $B$ bits/key, the **query time** is $O(1)$ and the **false positive probability** $\varepsilon$ is $\leq \frac{\ell}{2^{B-2}}$, where $\ell$ is the query range size.

- Optimal, according to the lower bound by Goswami et al. [SODA ’15]
- More succinct than their solution by more than 1 bit/key
- Works robustly out of the box; just specify $B$ or $\varepsilon$
Robust vs heuristic range filters

- Unlike Grafite, most range filters are not robust:
  - Performance depends on the choice of dataset and queries
  - A malicious user can issue adversarial queries
  - Still, they could provide some benefit in non-adversarial cases
- Can we design simpler heuristic range filters?
Bucketing: a simple heuristic range filter

1. Divide the universe \([0, u)\) into buckets of equal size \(s\)
2. Mark non-empty buckets with a compressed bit-vector
3. Solve queries by mapping ranges to bit-vector positions

Non-empty buckets: \[1\ 0\ 1\ 1\]
Bucketing: a simple heuristic range filter

1. Divide the universe $[0, u)$ into buckets of equal size $s$
2. Mark non-empty buckets with a compressed bit-vector
3. Solve queries by mapping ranges to bit-vector positions

Non-empty buckets:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Empty

$[a/s, b/s] = [a, b] \cap S = \emptyset$
Bucketing: a simple heuristic range filter

1. Divide the universe $[0, u)$ into buckets of equal size $s$
2. Mark non-empty buckets with a compressed bit-vector
3. Solve queries by mapping ranges to bit-vector positions

$[a, b] \cap S = \emptyset?$

Non-empty buckets: $1\ 0\ 1\ 1$
Experiments with heuristic range filters

- No filtering on \texttt{CORRELATED} queries, except for Proteus/REncoderSE:
  - Advantaged by auto-tuning
  - Very slow queries

- On other datasets, \texttt{Bucketing} offers:
  - FPR very close to the best-performing range filters
  - 6–13\times faster queries
  - 5–7\times faster construction

**Take-home message.** Heuristic range filters sacrifice robustness to work well on specific inputs, but there might be simpler solutions like \texttt{Bucketing}
Experiments with robust range filters

- **Space-vs-FPR.** Grafite is up to 4 and 5 orders of magnitude more effective than REncoder and Rosetta, respectively.

- **Query time.** Grafite is about 1 and 2 orders of magnitude faster than REncoder and Rosetta, respectively.

- **Construction time.** Grafite is up to 8 and 10 orders of magnitude faster than REncoder and Rosetta, respectively.

- **Take-home message.** If robustness guarantees are needed regardless of input data and future queries, Grafite is the range filter of choice.
Conclusion

• **Grafite** solves the lack of robustness in current practical range filters
  • Bounded FPR in optimal space
  • Efficient and predictable performance

• **Bucketing** simplifies the design of heuristic range filters
  • Almost equivalent FPR
  • Faster query and construction times

• **Future work**
  • Engineer a version of Grafite for string keys
  • Support insertions and study their impact on the FPR