Range filters, Adaptive filters

Yuvaraj Chesetti

February 2025

1 Range Filters

1.1 Motivation

Last lecture, we went over filters such as Bloom filters and quotient filters. A filter answers membership queries (Is x present in S) with false positives, which enable filters to use less space than an exact representation. Due to their compactness, filters are often stored in main memory to prevent negative queries from accessing the disk. These queries would have resulted in an empty result anyway, so its faster to skip the disk access and return an empty result immediately. Non-empty queries are never blocked, and few empty queries end up going to disk with probability ϵ, the false positive rate of the filter.

Range queries are common in applications, and it would be beneficial if we could filter out empty results for these queries. One example of a use case where range filtering would be really helpful are in Log-Structured-Merge (LSM) trees based key-value store. LSM trees shard data across multiple levels to maintain fast write performance, however range queries are particularly costly as they have to merge items from multiple levels. A range filter that filtered out empty range queries would improve the range-query performance of LSM trees.

1.2 Definition

A range filter generalizes a point filter for single-key membership - Is x in S -to range queries - Is there an x in S that lies between q_l and q_r?

A range filter F built on a data set S with n elements answers the range-emptiness queries of $[q_l, q_r] \cap S = \emptyset$ with maximum query length $R \geq q_r - q_l + 1$.

$$F([q_l, q_r], S) \rightarrow \begin{cases} \text{YES, } \exists x \in S, \text{ s.t } q_l \leq x \leq q_r, \text{else} \\ \text{YES, with } \Pr = \epsilon \\ \text{NO , with } \Pr = 1 - \epsilon \end{cases}$$

1.3 Space and query bounds

The simple way to implement a range filter is to just use any existing point filter design and individually query all items that lie in the range q_l and q_r.

If the false-positive rate of the underlying point filter is ϵ', the range-filter has a false-positive rate of $O(R\epsilon')$ by union bound. To obtain a false-positive rate of $\epsilon = R\epsilon'$, set $\epsilon' = \epsilon/R$.

The point filters we studied earlier have a space usage of $O(n \log_2(1/\epsilon'))$ bits. By setting $\epsilon' = \epsilon/R$, the range filter built using a point filter would use $O(n \log_2(R/\epsilon))$ bits. This filter also has query time $O(R)$.

Goswami et al. [4] show that this space usage is a lower bound, any filter needing to guarantee a false-positive rate of ϵ for queries at most length R, must use $\Omega(n \log_2(R/\epsilon))$ bits. However, they show that the query time can be improved to $O(1)$, and we will see a few filters that meet this query performance bound.

1.4 Prefix filters

Another simple design to implement a range filter is to divide the universe into partitions of size P, and for each key store $\lfloor x/P \rfloor$ in the point filter. If we choose P to be a power of 2, $P = 2^p$, this division operation can be done by masking the p lower order bits of the key.

To perform a range query $[q_l, q_r]$, we check if the point filter contains any key lying between $\lfloor q_l/P \rfloor$ and $\lfloor q_r/P \rfloor$. This has a query time of $O(R/P)$.

The problem here is that every time a partition is added to the filter, the filter assumes that all the items in the partition are part of the dataset. For workloads with empty queries close to keys present in the dataset, this filter will have a high false-positive rate. However, for some workloads, a simple heuristic like this could work well. For a detailed discussion, refer to the Bucketing filter [2].

1.5 Succinct Range Filter (SuRF)

The Succinct Range Filter (SuRF) [8] that stores the shortest prefix of a key that uniquely identifies it in a succinct trie representation [5]. To perform a range query $[q_l, q_r]$, the filter walks the trie to find the successor prefix of q_l, and returns a positive result if this prefix is less than or equal to q_r. To improve the false-positive rate, SuRF can also store additional information such as the hash of the suffix bits.

SuRF does not support updates as it stores the shortest unique prefix for every key, which must be recomputed on every new key. Additionally, the succinct encoding that SuRF uses to store the prefixes does not support updates. SuRF, like the prefix filter, cannot give a theoretical bound on its false-positive rate as it is dependent on the dataset and query workload.

1.6 Rosetta

Rosetta [6] is a range filter that maintains a bloom filter for all prefix lengths of a key. The topmost bloom filter is a bloom filter of prefixes of length 1, the next one is for prefixes of length 2, and so on. The lowest bloom filter stores the entire key and is essentially a point filter.

To perform a range query, Rosetta divides the query into dyadic intervals that share a common prefix. For example, for a range query between $[8, 12]$, the filter divides it into $[8, 11]$ and a single query for $[12]$. All keys in the range $[8, 11]$ have a binary representation of '10..' (assuming that all keys are 4 bits long).

For each interval, Rosetta checks the bloom filter corresponding to the common prefix length. If the filter returns an empty result, no keys with that prefix exist and the interval is empty. If the filter returns a positive result, there is a chance that this is a false positive. To lower the chances of a false positive, Rosetta performs an additional 'doubting' step in the filter at a level below it. The interval is returned as a positive result only if the lower filter also confirms that the interval is not empty. For instance, continuing the previous example, if the filter returned a positive result for the prefix '10..', the filter checks the lower level filter for the prefix '100.' and '101.'.

To guarantee a false-positive rate of ϵ, the lowest level filter bloom filter is built with a false-positive rate of ϵ, while all the higher level bloom filters are built with a false-positive rate of $1/(2 - \epsilon)$. The overall space usage of the Rosetta filter is $1.44 \log_2(R/\epsilon)$ bits, and queries are $O(\log_2(R))$. As the filter is bloom filter based, the filter does not support deletions.

1.7 Memento, a quotient filter for range queries

Memento [3] is a dynamic range filter based on the quotient filter. To provide the intuition for how memento works, we will divide the problem into a little-big universe problem.

1.7.1 Little-universe problem

In the little-universe problem, the goal is to build a single filter for m datasets $(S_1, S_2, ...S_m)$ containing keys from a universe u of size $|u| = P$. The single filter should be able to answer a range emptiness query for any of these datasets — for some S_i whether $[q_l, q_r] \cap S_i = \emptyset$.

We build the filter as an exact set that stores the key k if there exists a set S_i such that $k \in S_i$. The filter is essentially compressing all the m datasets into a single dataset. To answer a query $[q_l, q_r] \cap S_i = \emptyset$, the filter checks if its exact set contains a key in this range.

A query $[q_l, q_r] \cap S_i$ is a false positive if there exists another set S_j that contains items in this query range. Let us assume that the probability that two partitions has items in a range that the other does not has as ϵ_r, then the overall false-positive rate in the little universe problem can be bounded by $m\epsilon_r$ by union bound.

1.7.2 Big-universe problem

Finally, to solve the big universe problem, Memento divides the larger universe into U into multiple partitions of size P, such that P_i covers the range $[iP,\ (i+1)P]$. The partitions are then assigned to one of M little-universe problems $(u_1, u_2, ...u_M)$ using a hash function. Here, the filter is essentially scattering all the partitions randomly into one of the M multiple little universes. To answer a range query, the query is then decomposed into multiple little-universe problems by dividing into intervals that lie completely inside a partition.

1.7.3 False-Positive rate and space usage

To answer range queries of max query length R on a dataset of n items, Memento sets the partition size P to be the size of the maximum query length R, and divides the big universe into $n\epsilon/P$ little universe problems. There are at most n partitions (one for each item) that need to be split across $n\epsilon/P$ little-universe problems. The average expected number of partitions in a single little-universe is thus ϵ. The overall false-positive rate of a single little-universe problem is $\epsilon\epsilon_r$, where ϵ_r is the probability that two partitions in the little-universe contain one partition causes a false-positive in the other partition. This probability depends on the data distribution, but even if we assume it to be 1 the overall false-positive rate can be upper bounded by ϵ.

The filter uses $O(\log_2(1/\epsilon) + \log_2(R))$ bits per item in a quotient filter. The first set of bits $O(\log_2(1/\epsilon))$ correspond to the big-universe to little-universe mapping, while the $\log_2(R)$ bits correspond to storing the item in the little universe.

2 Adaptive Filters

2.1 Motivation

In all the filters we've seen so far, once the filter is built, its false-positive set is fixed. If a query for an item k is a false positive, querying the item again will still yield a false positive. This can make the filter susceptible to skewed or adversarial workloads.

The problem here is that the false-positive rate ϵ is *static*, it is the probability that a single random query is a false positive, given that it is actually an empty query. The static false-positive rate does not guarantee the probability that an empty query is a false positive if ϵ if it is issued by a user who has recorded the result of arbitrary number of previous queries. If the user is adversarial, that is, their aim is to drive the observed false-positive rate of the filter to 1, they can simply repeat the first encountered false positive over and over.

Ideally, we would like to have a filter that has a *sustained* false-positive rate of ϵ, the probability that an empty query is false positive is ϵ, independent of the query workload.

A filter can guarantee a *sustained* false-positive rate of ϵ only if it is *adaptive*, the filter is able to learn from and correct from its mistakes.

2.2 Definition and bounds

Bender et al. [1] define a filter as *strongly adaptive* if it guarantees that the number of false positives after n queries is $O(\epsilon.n)$ with high probability.

Wen et al. [7] define an even stronger notion of adaptivity called *monotonically adaptive*. In addition to being strongly adaptive, the monotonically adaptive filter never forgets a false positive that it has seen before.

Bender et al. show that to support a filter being strongly adaptive, it requires atleast $O(n)$ bytes of space - it must store an exact representation of the keys. This does not prevent the filter from being practical, as the space can be split between a small structure kept in main memory, and a larger structure kept on disk and accessed only on updates.

2.3 Adaptive quotient filters

This section gives a high level overview of the adaptive quotient filter designed and implemented by Wen et al. [7].

2.3.1 Adaptive filter setup

An adaptive filter can be implemented by splitting the filter into two parts — a small in-memory structure that is accessed on queries, and a larger auxiliary structure that is kept on disk and ideally accessed only on updates.

An adaptive filter also requires feedback, when the system detects that the filter has returned a false positive for a query k, the system must inform the filter so that it can adapt to this query.

2.3.2 False positives in quotient filters

False positives in an quotient filter occur due to fingerprint collisions. Recall that the quotient filter stores an exact set of fingerprints $\{h_f(x), x \in S\}$, where S is the dataset and every fingerprint $h_f(x)$ is f bits long. A query for an item k is a false positive if $k \notin S$, but there exists a key x such that $h_f(k) = h_f(x)$. The high-level idea then of implementing adaptivity in a quotient filter is to resolve this fingerprint collision in the filter.

2.3.3 Resolving false positives through variable-length fingerprints

The adaptive quotient filter resolves fingerprint collisions by storing *variable-length* fingerprints. Initially, when the filter is constructed, the filter stores f length fingerprints for each key. This fingerprint is generated by using a hash function that generates a long hash h, and taking the f most-significant bits of the hash.

When the filter first encounters a false positive for a key k, this means that there exists another key $x \in S$ that shares the same first f bits of their hash i.e., $h(x)$ and $h(k)$ have the same f length prefix in their bit string. To resolve this collision, the fingerprint of the key x is extended by e bits so that the first $f + e$ bits of $h(x)$ and $h(k)$ no longer match, and the filter is updated with the longer fingerprint. In expectation, $e = 2$ bits are enough to break the collision.

To perform a query for a key k, the quotient filter searches for a variable-length fingerprint that is a prefix of $h(k)$. The quotient filter also maintains the invariant that the fingerprints inside the quotient filter are not prefixes of each other, thus the fingerprint for key k can match with at most one fingerprint in the filter.

The quotient filter encodes the variable length fingerprints with one additional metadata bit compared to the non-adaptive quotient filter, and uses the fast rank-and-select operations to implement the search for matching variable-length fingerprints.

2.3.4 Reverse map

Queries in an adaptive quotient filter do not access the disk; it is the similar lookup process followed by the non-adaptive quotient filter.

When the filter returns a false positive, it is not detected until the system goes to disk and fetches an empty result. The system then informs the filter that a false positive has occurred.

In the adaptation process described above, the filter only has access to the variable-length fingerprint derived from $h(x)$ that caused the false positive. To add more bits to the fingerprint, the filter requires the original key x that cannot be derived from the hash.

This is where the auxiliary data structure that the filter stores on disk comes into play. The auxiliary data structure is a reverse mapping of all variable-length fingerprints stored on disk to the keys that map to it. This is $O(n)$ bytes, as it requires storing the original key.

The adaptive filter only introduce disk accesses to inserts (to update the reverse map) and when adapting to false positives(to fetch the key and update the reverse map with extended fingerprint), and is not needed for queries. Inserts and false positives Thus, the adaptive filter thus only introduces disk accesses to execution paths that would have accessed the disk anyways — inserts into the filter are usually coupled with an insert into the database on disk, while false-positives would have resulted in a disk access to fetch the data.

References

[1] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *Proc. 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 182–193, Paris, France, October 2018.

[2] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. Grafite: Taming adversarial queries with optimal range filters. *Proc. ACM Manag. Data*, 2(1):3:1–3:23, 2024.

[3] Navid Eslami and Niv Dayan. Memento filter: A fast, dynamic, and robust range filter, 2024.

[4] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms*, pages 769–775, 2015.

[5] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[6] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2071–2086, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Richard Wen, Hunter McCoy, David Tench, Guido Tagliavini, Michael A. Bender, Alex Conway, Martin Farach-Colton, Rob Johnson, and Prashant Pandey. Adaptive quotient filters. *Proc. ACM Manag. Data*, 2(4), September 2024.

[8] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 323–336. ACM, 2018.