Space efficient indexes for the big data era
Sidirourgos, L.

Citation for published version (APA):
Sidirourgos, E. (2014). Space efficient indexes for the big data era

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

Memory Efficient Bloom Filters for Skewed Access Patterns

Main memories are becoming sufficiently large that most OLTP databases can fit entirely in memory. Nevertheless, OLTP workloads often exhibit skewed access patterns where some records are hot (frequently accessed) and many records are cold (infrequently or never accessed). It is still more economical to store the cold records on secondary storage. To minimize unnecessary disk accesses an in-memory Bloom filter can be used to quickly check whether a given record exists in cold storage. In this chapter, we show how skew in access patterns can be exploited to improve Bloom filter efficiency (less space and/or lower false positive rate). We do this by splitting the filter into many smaller ones, where each filter covers only a small subset of records. The challenge is, given the (skewed) access frequencies per subset of records, to determine the optimal number of bits per Bloom filter. We define a mathematical model and show how to optimally size the Bloom filters. We also describe how to adjust the sizing of the filters because of changes in access patterns or deterioration caused by updates. Furthermore, since with this split the Bloom filters are smaller, it is faster to create and query them because of fewer cache misses. An extensive experimental evaluation confirms that our algorithms construct better Bloom filters optimized for skewed access patterns, that they achieve a lower false positive rate or lower memory consumption depending on the settings, and that they can adapt gracefully to data and workload changes.
3.1 Motivation

Database systems have traditionally been designed assuming that data is disk resident. However, the drop in memory prices over the past 30 years is invalidating this assumption. Several database engines have emerged that optimize for the case when most data fits in memory [30, 37, 63, 77, 78]. This architectural change necessitates a rethink of all layers of the system, from concurrency control [47] and access methods [66, 72] to query processing [11].

In OLTP workloads record accesses tend to be skewed. Some records are “hot” and accessed frequently while others are “cold” and accessed infrequently. Clearly, good performance depends on the hot records residing in memory but cold records can be moved to cheaper external storage such as flash with little effect on overall system performance.

Consider, for example, a table containing customer orders. The most recent, active orders are stored in an in-memory table – the hot table – with two hash indexes, one on order number and one on customer number. Older, closed orders are accessed rarely and, to reduce memory requirements and overall system cost, are moved to another table – the cold table – stored on a flash SSD. The cold table also has hash indexes on order number and on customer number. Suppose orders that have been closed for more than 30 days are automatically moved to the cold table.

To retrieve all orders for a customer we must check both the hot table and the cold table. For customers that have no orders older than 30 days, accessing the cold table is wasted effort. A typical way to avoid such unnecessary accesses is to use an in-memory Bloom filter, in this case, one on customer number that covers customers with orders in the cold table. Before accessing the cold table, we do a quick check against the Bloom filter and if the result is negative, we have avoided an expensive lookup in the cold table.

3.1.1 Contributions

Instead of a single large Bloom filter per index, we propose to use a collection of smaller Bloom filters, called hereafter Split Bloom filters. Each smaller filter covers a fixed set of hash buckets (in the case of a hash index) or a fixed set of pages (in the case of a B+tree index). Each filter may, for example, cover two consecutive buckets (pages). The split Bloom filters are of variable size, where more bits are allocated to buckets (pages) that contain many distinct keys and/or are frequently accessed. Using a collection of smaller Bloom filters is a simple idea but it offers several advantages.

- We can exploit skew by allocating more bits to filters that are frequently accessed
and/or cover larger subsets of records. We have developed an algorithm (section 3.4) to determine the optimal allocation of bits to filters. Our experiments confirm that the split Bloom filters are more efficient (less space and/or lower false positive rate) than a single large Bloom filter.

- It is faster to build a collection of split filters than a single large filter and checking a small filter is faster because of fewer cache misses. This is confirmed by our experiments.

- When the set of records covered by a filter changes due to insertions and deletions, the filter no longer performs optimally (too many bits set). When this occur we can remedy the situation simply be rebuilding the filter. This is cheaper than rebuilding a single large Bloom filter, because each split filter covers only a small subset of records.

- Split Bloom filters also make it possible to adapt to changes in access frequencies. If a split filter becomes more frequently accessed and/or produces more false positives than expected, we can rebuild the filter and increase its size. Vice versa, if a filter is hardly ever accessed we can rebuild it and reduce its size, thereby saving memory space. How to detect and adapt to changing access frequencies is discussed in section 3.5.

This work is part of a research project investigating how to manage cold data in Hekaton, an OLTP database engine optimized for main-memory and integrated into SQL Server [17, 26]. The goal is to enable Hekaton to automatically migrate cold records to cheaper secondary storage while still providing access to such records transparently. Needless to say, it is important to avoid unnecessary accesses to secondary storage. We considered using classical Bloom filters but found them too inflexible. We also considered counting Bloom filters but deemed them too space inefficient and not able to exploit skew.

### 3.1.2 Outline

The rest of this chapter is organized as follows. Next section summarizes related work on Bloom filters. Section 3.3 presents the system infrastructure. Section 3.4 details the process of splitting Bloom filters. In Section 3.5 we study the different cases that call for a total or partial rebuild of Split Bloom filters. The experimental evaluation of the proposed algorithms is presented in Section 3.6. We conclude our work with Section 3.7.
3.2 Related Work on Bloom Filters

Bloom filters were introduced in 1970 by Burton H. Bloom [12]. Since then, numerous extensions to the original design have been proposed. One of the most notable extensions are spectral Bloom filters [23] which use counters instead of single bits and can support multi-sets. Instead of only setting a bit whenever an element is hashed to a position, the spectral Bloom filters increase a counter associated with that position, thus allowing correct support of both count queries and deletion of elements from the filter.

The Bloomier filter [21] is an extension of Bloom filters that supports membership checks over static functions that divide the domain into several subsets. Given an element, it decides if it belongs to set $A, B, C, \ldots$ or none of them. To achieve this, multiple Bloom filters are created for each subset, hence it performs better if the number of subsets is small. Compressed Bloom filters [57] study the problem of optimizing the compression rate of a Bloom filter instead of only optimizing the false positive ratio. The author demonstrates that it is better to use fewer hash functions to achieve better compression rates since there will be more unset bits.

Several other variants of Bloom filters aimed at making them more space and time efficient have been proposed in the literature [64, 69]. In [69] the authors propose a cache efficient variant that uses blocks of bits and precomputed bit patterns to be set. In [43] only two hash functions are used to produce the $k$ needed, resulting in less cpu time. The false positive ratio can be significantly reduced by using partitioned hashing [31]. This is done by first defining many different families of hash functions, and then hashing an element to determine to which family it belongs. These functions are then used as in the traditional Bloom filters.

Scalable Bloom filters [3] address the problem that arises if the number of elements to be inserted in a Bloom filter is unknown, yet a strict false positive ratio needs to be enforced. The authors proposed a scalable growth of the Bloom filter by appending bits as needed.

Weighted Bloom filters [16] and Conscious Bloom filters [89] have both been proposed to deal with skewed workloads. In Weighted Bloom filters, the authors propose to use different number of hash functions for each key depending on the frequency it is looked up and the likelihood to be a member, instead of decreasing or increasing the number of bits used, as per our solution. Similarly, the authors of Conscious Bloom filters present a $(2 + \epsilon)$-Approximation algorithm to decide how many hashes to use. Weighted and Conscious Bloom filter do not facilitate updates or adjusting to changes in the access skew, since they use a single large filter instead of many smaller ones that are easier to maintain.
3.3 Avoiding Trips to Cold Storage

For each table that resides in memory, there is a corresponding table in the cold store for records that are infrequently accessed. Records are never duplicated – a record is either in the hot store or in the cold store. A cold table may have any number of indexes, each defined for a different search key of the records. A record may be accessed through any index so we need multiple filters, one for each index.

A straightforward approach is to build a single large Bloom filter that covers an entire index of the cold table. The filter would use a predefined number \( m \) of bits, called hereafter the memory budget, and \( k \) hash functions to identify which bits to be set. A Bloom filter performs optimally when the number of false positives is minimized. It has been shown [12] that the optimal balance between the number of hash functions \( k \), the number of bits \( m \) and the number of distinct keys \( n \) is achieved when \( k = \ln 2^{m/n} \).

Here, instead of maintaining a single large Bloom filter per index, we split it into a collection of smaller Bloom filters that add up to the same memory budget \( m \). Each of the smaller Bloom filters may be of different size. The number of bits for a filter is determined based on i) how frequently the records covered by the filter are accessed, and ii) the number of (distinct) keys hashed. The intuition is to assign more bits to Bloom filters that contain more keys and/or are queried more often, which is where the bits yield the most benefits. There is little benefit in using a large filter for a small set of keys or for keys that are hardly ever queried, because even if it returns a false positive half of the time, the overall impact on the false positive rate of the entire index is negligible.

Obviously, our approach benefits from skewed access patterns; access skew provides opportunities to redistribute the memory budget so as to achieve a lower false positive rate than a single large Bloom filter. Aging data, where the older a record becomes the less likely it is to be queried, exhibits such skew and this is a fairly common pattern.

Before presenting our mathematical model and the algorithms for optimal sizing each Split Bloom filter, we detail the over all system design for two different types of indexes, namely hash indexes and B+trees.

3.3.1 Split Bloom Filters for Hash Indexes

Consider an in-memory table with \( n \) records of customer orders where the column \( order_id \) is a unique key and the column \( customer \) identifies the customer that placed the order, as illustrated in Figure 3.1. Older orders are moved to the cold store and indexed on \( customer \) using a hash index. Given a query that retrieves all orders placed by a customer, we first search the in-memory hot table, possibly using a hash index on
CHAPTER 3. MEMORY EFFICIENT BLOOM FILTERS

Figure 3.1: In-memory Split Bloom filters for a cold store resident hashfile index.

customer. Next, we scan the appropriate bucket of the hash index in the cold store to find the customer’s older orders.

To avoid an unnecessary access to the cold store, we cover a predefined number of hash buckets in the cold store with a smaller Bloom filter. The choice of how many hash buckets to cover per filter is a space versus efficiency trade-off. Smaller filters covering fewer buckets are easier to maintain, while fewer larger filters reduce management overhead. In Figure 3.1 there are four different hash buckets in the cold store. Suppose we split the Bloom filter into four smaller ones each covering one hash bucket. Given the customer, we first compute the hash value of the key to determine to which hash bucket it corresponds to in the cold store. But, before accessing the cold store, we probe the corresponding Bloom filter, i.e., the one that covers the target hash bucket, to determine whether a trip to the cold store can be avoided.

3.3.2 Split Bloom Filters for B+Tree Indexes

Consider an index that supports ranges, such as BW-trees [49], a novel lock-free version of in-memory B+trees. Such a B+tree variation index over the customer column of the orders table is shown in Figure 3.2. A query about all orders of a customer will...
3.3. AVOIDING TRIPS TO COLD STORAGE

In−Memory B+tree

Split Bloom Filters For B+tree

Figure 3.2: In-memory Split Bloom filters for a B+tree index.

traverse the B+tree until it reaches the leaf nodes that point to the records in the orders table. The next step is to retrieve the corresponding pages in the cold store with the older orders of the same customer. To make use of Split Bloom filters and avoid unnecessary access of pages in the cold store, we add to the leaf nodes a field, \( \text{filter\_id} \), that identifies which Split Bloom Filter covers the node’s key range. Multiple adjacent leaf nodes may share the same filter, in particular, leaf nodes covering key ranges with few records in the cold store. For leaf nodes with no records in the cold store \( \text{filter\_id} \) is set to null. If \( \text{filter\_id} \) is set, we locate the corresponding Bloom filter and check whether it is necessary to access the cold store.

3.3.3 Split Bloom Filters Construction

We store Split Bloom filters in a regular in-memory table to take advantage of the system’s transactional guarantees. This ensures that the filters will be durable and consistent with the data in the cold store. We use one Split Bloom filter table per cold table, thus storing in this table all filters for all indexes of that particular cold table. Therefore, for each cold table, there is one Split Bloom filter table in memory. One could also use one Bloom filter table per index, per database, or even a single system-wide table.

An example of a Split Bloom table is shown in Figure 3.3. Each Bloom filter is identified by a unique combination of the identifier of the index, which is given by the catalog information of the system, and the id of the Split Bloom filter. For example, the first record with identifier 1.1 of the Split Bloom filter table in Figure 3.3 corresponds
CHAPTER 3. MEMORY EFFICIENT BLOOM FILTERS

<table>
<thead>
<tr>
<th>index.filter</th>
<th>n</th>
<th>k</th>
<th>m</th>
<th>f</th>
<th>cnts</th>
<th>Bloom filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>n_{1,1}</td>
<td>k_{1,1}</td>
<td>3</td>
<td>f_{1,1}</td>
<td>...</td>
<td>110</td>
</tr>
<tr>
<td>1.2</td>
<td>n_{1,2}</td>
<td>k_{1,2}</td>
<td>9</td>
<td>f_{1,2}</td>
<td>...</td>
<td>100111010</td>
</tr>
<tr>
<td>1.3</td>
<td>n_{1,3}</td>
<td>k_{1,3}</td>
<td>4</td>
<td>f_{1,3}</td>
<td>...</td>
<td>1010</td>
</tr>
<tr>
<td>1.4</td>
<td>n_{1,4}</td>
<td>k_{1,4}</td>
<td>6</td>
<td>f_{1,4}</td>
<td>...</td>
<td>100110</td>
</tr>
<tr>
<td>2.1</td>
<td>n_{2,1}</td>
<td>k_{2,1}</td>
<td>9</td>
<td>f_{2,1}</td>
<td>...</td>
<td>100101110</td>
</tr>
<tr>
<td>2.2</td>
<td>n_{2,2}</td>
<td>k_{2,2}</td>
<td>4</td>
<td>f_{2,2}</td>
<td>...</td>
<td>1100</td>
</tr>
<tr>
<td>2.3</td>
<td>n_{2,3}</td>
<td>k_{2,3}</td>
<td>6</td>
<td>f_{2,3}</td>
<td>...</td>
<td>101010</td>
</tr>
</tbody>
</table>

Figure 3.3: In-memory Split Bloom Filters table.

to the hash index with id=1 and the Split Bloom filter with id=1. The id of the Split Bloom filter is identified as follows. For a hash index, each Split Bloom filter covers a fixed number of hash buckets, say \( w \) buckets, so the Split Bloom filter id of a bucket \( b \) is simply \( \lfloor b/w \rfloor \). In the case of a B+tree, the id is given by the leaf node. The next three attributes of the Split Bloom Table in Figure 3.3 are the number \( n_i \) of distinct records in the group of buckets (or group of pages) covered by filter \( i \), \( k_i \) the number of hash functions used by that Split Bloom filter, and \( m_i \) the number of bits used. The counter \( f_i \) is incremented whenever the Split Bloom filter is probed and enables us to estimate the filter’s access frequency. Next, there are a number of attributes, denoted as cnts in Figure 3.3. The details of these counters are given in Section 3.5 where we discuss Split Bloom filters maintenance strategies. Finally, the last attribute of the Split Bloom filter table is a variable length bit vector that stores the actual Bloom filter.

The main function for creating the Split Bloom filters over an index of the cold table is detailed in Algorithm ???. It simply scans the buckets or pages covered by each Split Bloom filter, hashes the keys and sets the selected bits in the filter, and finally creates and inserts a record with the filter in the Split Bloom filter table. In the next section we determine the parameters to optimally size each Split Bloom filter according to the frequency and the number of distinct keys it covers.

### 3.4 Optimal Sizing of Split Bloom Filters

We wish to split a memory budget of \( m \) bits into \( l \) Bloom filters such that the total false positive ratio is minimized. Denote the size (in bits) of Bloom filter \( i \) by \( m_i \). The sum
Algorithm 3.1 Split Bloom filters Construction

\[ l = \text{totalBuckets/bucketsPerGroup}; \]
\[ (m[], k[]) = \text{Algorithm 3.2}(l, m, n, n[], f[]); \]

\textbf{for} \( i \leftarrow 1 \) to \( l \)
\hspace{1em} /* allocate a new Bloom filter of size \( m[i] \) */
\hspace{2em} Bloomfilter = allocate\_bits\( (m[i]) \);
\hspace{2em} \textbf{for} key in buckets \( i\times\text{bucketsPerGroup} \) \textbf{until} \( (i + 1)\times\text{bucketsPerGroup} \)
\hspace{3em} \text{hash}_{h}(\text{key}) \rightarrow \text{Bloomfilter};
\hspace{2em} \textbf{end for}
\hspace{2em} \textbf{end for}

\( \text{id} = (\text{IndexId} << 32) | i; \)
\( \text{new record}(\text{id}, n[i], k[i], m[i], f[i], \text{Bloomfilter}); \)
\( \text{insert}(\text{record}) \rightarrow \text{SplitBloomTable}; \)
\textbf{end for}

of bits assigned to the filters should equal the memory budget \( m \), that is,
\[ m = \sum_{i=1}^{l} m_i. \]

The false positive ratio of the \( i \)th Bloom filter can be approximated by
\[ p_i = (1 - e^{-k_i n_i / m_i})^{k_i}, \tag{3.1} \]
where \( k_i \) is the number of hash functions used, and \( n_i \) the number of distinct keys included in the \( i \)th Bloom filter.

To be able to estimate the access frequencies of the filters, each filter has a counter \( f_i \) that is incremented on every access to the filter. Let \( f \) denote the sum over all access counters,
\[ f = \sum_{i=1}^{l} f_i. \]

The total or \textit{overall} false positive ratio \( P \) for the \( l \) Bloom filters covering an index is equal to
\[ P = \sum_{i=1}^{l} \frac{f_i}{f} p_i. \tag{3.2} \]
CHAPTER 3. MEMORY EFFICIENT BLOOM FILTERS

That is, the overall false positive rate of a collection of Bloom filters is a weighted average of the false positive ratios of the individual Bloom filters, weighted with their respective fraction of total accesses.

To minimize $P$, optimal values for all $m_i, i = 1, \ldots, l$ must be determined. The problem amounts to a non-linear minimization problem that can be solved using Lagrange multipliers. Our derivation of the solution is included at the end of this chapter. We derive the following formula for finding the optimal filter size $m_i$ for $i = 1, \ldots, l$

$$m_i = (\ln^{-2} 2 \gamma_i + \frac{m}{n}) n_i$$  \hspace{1cm} (3.3)

where

$$\gamma_i = \ln f_i - \ln n_i + \sigma$$ \hspace{1cm} (3.4)

$$\sigma = \frac{1}{n} \sum_{i=1}^{l} n_i (\ln n_i - \ln f_i).$$ \hspace{1cm} (3.5)

The optimal size of a Bloom filter is a function of the number of keys $n_i$ and a factor $\gamma_i$. $\gamma_i$ is equal to the difference between the natural logarithm of the frequency $f_i$ and the keys $n_i$ plus a constant $\sigma$. $\sigma$ is the same for all Bloom filters, thus it needs to be computed only once. Once $m_i$ is determined, then $k_i$ is set to $k_i = \ln 2^{\frac{m_i}{n_i}}$.

Algorithm 3.2 implements the mathematical formulas derived here. The input to the algorithm consists of the number of filters $l$, the memory budget in bits $m$, the total number $n$ of keys, as well as two arrays, namely $n[]$ with the number of distinct keys per group of hash buckets or group of B+tree pages, and $f[]$ containing the access frequencies.

For the initial build of the Bloom filters the number of distinct keys, $n[]$, can be computed by a scan over the index. If estimates of the frequencies $f[]$ are not available we assume uniform access frequencies.

Algorithm 3.2 starts by first computing $\sigma$. For this, $l$ multiplications and subtractions are needed, $2 \times l$ calls to function $\log$, and finally one division. Then, for each Bloom filter the parameter $\gamma$ is computed. The cost for computing $\gamma$ is $2 \times l$ calls to function $\log$, one subtraction and one addition. If $\gamma$ is zero or negative, then the access frequency is so rare, or the number of unique keys are so few, that no bits need to be used. In this case, a query to this filter will always return true, and a trip to the cold table will not be avoided. If parameter $\gamma$ is a positive number, the number of bits and hash functions per Bloom filter are computed.

Our model is a continues approximation and thus produces non-integer (i.e., floating point) values instead of integer values for $m[i]$. Therefore, we use function ceil
Algorithm 3.2 Sizing Split Bloom filters

**Input:** number of filters \( l \), total bits to be used \( m \), total keys \( n \), keys per filter \( n[] \), access frequencies per filter \( f[] \)

**Output:** number of bits per filter \( m[] \), number of hash functions per filter \( k[] \)

// first compute \( \sigma \)
float \( \sigma = 0 \);
for \( i \leftarrow 1 \) to \( l \)
   \( \sigma = \sigma + n[i] \ast (\log(n[i]) - \log(f[i])); \)
end for

\( \sigma = \sigma/n; \)

// next calculate the number of bits
\( \log2 = 1.0/(\log(2.0) \ast \log(2.0)); \)
for \( i \leftarrow 1 \) to \( l \)
   \( \gamma = \log(f[i]) - \log(n[i]) + \sigma; \)
   if (\( \gamma < 0 \))
      \( m[i] = 0; \)
      \( k[i] = 0; \)
   else
      \( m[i] = \text{ceil}((\log2 \ast \gamma + (m/n)) \ast n[i]); \)
      \( k1 = \text{floor}(\log2 \ast m[i]/n[i]); \)
      \( k2 = \text{ceil}(\log2 \ast m[i]/n[i]); \)
      \( k[i] = \text{min}(fp(k1), fp(k2)); \)
   end if
end for
to take the closest integer that is larger than \( m[i] \). However, to decide the number of hash function, both options of \( \lfloor k[i] \rfloor \) and \( \lceil k[i] \rceil \) are checked and the one that produces the smaller false positive ratio \( p \) is used. The theoretical false positive ratio can never be exactly achieved, but only approximated by any Bloom filter. This is because of the natural limitation of not being able to use a non-integer number of bits or a fraction of hash functions as the formulas indicate.

### 3.5 Split Bloom Filter Maintenance

To maintain a low false positive rate it may be necessary to rebuild a few or all filters. In this section we describe our approach to monitoring filter performance and triggering filter maintenance.

#### 3.5.1 Incremental Build of Individual Filters

The algorithms presented in Section 3.4 derive the optimal size for each Bloom filter given the number of keys, the number of accesses, and the available memory budget. However, for reasons of efficiency it is preferred to rebuild the Bloom filter that needs to, instead of rebuilding the entire collection of Split Bloom filters. The incremental build procedure described in this section ensures that the target false positive ratio initially set by Algorithm 3.2 is achieved, even if the access frequency or the number of unique keys have changed for the specific filter to be rebuild. Experiments detailed in Section 3.6 show that by rebuilding filters one at a time we still approach the global minimum as computed by Equation 3.3.

Each filter contributes to the overall false positive by a fraction \( f_i \) of its own false positive. If that fraction changes because of access skew, or the false positive ratio \( p_i \) changes because of updates, then an incremental rebuild must be performed. However, the following equation must hold in order for the overall contribution to remain the same

\[
\frac{f_i}{f} p_i = \frac{f'_i}{f'} p'_i
\]  

(3.6)

where \( f'_i, f' \) are the new values of access frequency that triggered the incremental rebuild. This observation leads to the following equation for computing the new size of a single Bloom filter

\[
m'_i = -n'_i \ln \left( \frac{f'_i}{f'_i f} p'_i \right) \ln^{-2} 2. \]  

(3.7)
3.5. SPLIT BLOOM FILTER MAINTENANCE

Algorithm 3.3 Incremental sizing of a Bloom filter

Input: record(nᵢ, kᵢ, mᵢ, fᵢ) of filter i to build, FPtᵢ current false positive target of filter i, Frᵢ frequency ratio of filter i

Output: a new record with updated Bloom filter, updates FPtᵢ and Frᵢ

\[ \log_2 = 1.0/(\log(2.0) \ast \log(2.0)); \]
// calculate the new false positive target
\[ \text{new}_\text{fp} = (f/fᵢ) \times Frᵢ \times FPtᵢ; \]
if (new_fp > 1)
    new_fp = 1;
    mᵢ = 0;
    kᵢ = 0;
else
    mᵢ = -nᵢ \times \log_2 \times \log(\text{new}_\text{fp});
    k₁ = \text{floor}(\log(2.0) \ast mᵢ/nᵢ);
    k₂ = \text{ceil}(\log(2.0) \ast mᵢ/nᵢ);
    kᵢ = \text{min}(fp(k₁), fp(k₂));
endif
Bloomfilter = rebuildFilter(i, mᵢ, kᵢ);
FPtᵢ = new_fp;
Frᵢ = fᵢ/f;
return new record(nᵢ, kᵢ, mᵢ, fᵢ, Bloomfilter), FPtᵢ, Frᵢ;
The details of the derivation of this formula are described at the end of this Chapter. Note that $n'_i$ is the updated number of unique keys, thus taking into account any insertion or deletions.

Algorithm 3.3 implements the above formula to incrementally rebuild a filter. The input is the record with the counters and the Bloom filter to be rebuild, as well as counters $FPt_i$ and $Fr_i$. $FPt_i$ is exactly the $p_i$ of Equation 3.7, and $Fr_i$ is the fraction $\frac{f_i}{f}$. While the current $f'_i$ is the counter $f_i$ stored in the record of the Bloom filter. The first step for Algorithm 3.3 is to compute the new false positive ratio $p'_i$. Next, if it is smaller than 1.0, the number of bits $m_i$ are computed from the formula. If the new false positive is greater than 1.0, the access frequency is so low that no filter is needed. Finally, the algorithm will rebuild the filter with the newly computed parameters and replace the old record in the Bloom table.

During the incremental rebuild, we only scan the hash buckets or B+tree pages that are covered by the specific filter, and not the entire index. Therefore it is more efficient than a total rebuild. In the experiments we will show that a series of incremental rebuilds approach the same overall false positive ratio that would have been achieved, if we performed a total rebuild of the collection of Bloom filters.

### 3.5.2 How Bloom Filters Deteriorate

#### Inserts

When new records are added to the cold store, some Bloom filters may have to be updated to prevent false negatives. False negatives occur when a Bloom filter incorrectly indicates that a requested key does not exist in the cold store even though it does. This produces an incorrect query result which, of course, is unacceptable. The filter update has to be done in a manner that ensures that false negatives cannot occur at any point. We store Bloom filters as a variable length attribute of an in-memory table and take advantage of the transactional infrastructure of the database engine. The filter update is done as part of the same transaction that updates the cold store so the Bloom filter update commits with the cold store update.

#### Deletes

When a record $r_1$ is deleted, it is not safe to turn off the bits set by $r_1$ in a Bloom filter because some other record $r_2$ still in the cold store may have set one or more of the same bits. Turning off the bits would “hide” $r_2$ and cause a false negative. To ensure correctness we do not update Bloom filters when a record is deleted. This may, however, increase the false positive rate because the Bloom filter still indicates that $r_1$
is present. Deletes, same as inserts, will cause Bloom filters to deteriorate and produce a higher false positive rate than necessary. Either too many bits will be set because of insertions or too many bits will remain set despite records having been deleted. At some point the Bloom filters have to be rebuilt in order to ensure optimal performance.

Changes in Access Frequency

Changes in access frequencies may also cause suboptimal performance and call for a rebuild. Suppose the access frequency of a hash bucket or a B+tree page decreases dramatically. If so, the contribution of the Bloom filter covering these buckets (pages) to the total false positive rate is also reduced. The Bloom filter may now be unnecessarily large; it may be better to shrink it and invest the freed up bits elsewhere. Similarly, if the access frequency for a bucket (page) increases, its covering Bloom filter may be too small for optimal performance.

Skewed Access Patterns

There is an additional situation that may call for a rebuild. The formulas for estimating false positive rates and sizing Bloom filters assume that incoming requests select bits at random. In practice that may not be the case. Suppose we have sized a Bloom filter for an expected false positive rate of 5%. Now consider a particular key value $K$ that results in a false positive. If all incoming requests are for key value $K$, the actual false positive rate will be 100%. This is an unlikely case but it shows that the actual distribution of requests may result in false positive rate that is higher (or lower) than theoretically expected. If the Bloom filter is rebuild with a different size, a different set of bits will be set and chances are that $K$ will no longer cause a false positive.

When a set of Bloom filters have deteriorated sufficiently we rebuild them, possibly also changing their size. A rebuild requires scanning the buckets (pages) covered by the filters, recomputing the filters, and updating the Bloom table with the new filters. This has to be done in a transactional manner to ensure that false negatives cannot occur. Apart from rebuilding a set of filters, one may also incrementally build individual Bloom filters such that the over all false positive target of the collection remains fixed. The process of incrementally building filters, instead of total rebuild, is described later in this section. First we detail the counters maintained and the policies used for triggering a rebuild.
3.5.3 Counters

To be able to monitor Bloom filter performance and react to changes we maintain several counters at different levels. There are six counters for each Bloom filter.

- $I_{idx,i}$, the number of new distinct keys inserted into buckets (pages) covered by filter $i$ of index $idx$,
- $D_{idx,i}$, the number of distinct keys deleted from buckets (pages) covered by filter $i$ of index $idx$,
- $N_{l_{idx,i}}$, the number of lookups in filter $i$ of index $idx$,
- $N_{f_{idx,i}}$, the number of false positives produced by filter $i$ of index $idx$,
- $FP_{t_{idx,i}}$, the theoretical – or target – false positive ratio of filter $i$ as computed during the last rebuild of the entire collection of Bloom filters for index $idx$, or during the last incremental build of filter $i$, and
- $Fr_{idx,i}$ the access frequency fraction $\frac{L}{F}$ of filter $i$ as noted during the last rebuild or incremental build.

Counters $I_{idx,i}$ and $D_{idx,i}$ are used to determine if a filter rebuild is needed because of too many updates. $N_{l_{idx,i}}$, $N_{f_{idx,i}}$ can be used to continuously monitor the real false positive ratio and compare it with the target ratio $FP_{t_{idx,i}}$ so as to be able to detect large deviations between observed and theoretical false positive. In addition, $FP_{t_{idx,i}}$ and $Fr_{idx,i}$ are used for incremental rebuilds.

There are four counters for the collection of Bloom filters covering an index.

- $N_{l_{idx}}$, the number of lookups in index $idx$,
- $N_{f_{idx}}$, the number of false positives produced by the filters of index $idx$,
- $E_{idx}$, the difference between memory used and the memory budget for the filters of index $idx$, and
- $FP_{t_{idx}}$, the theoretical – or target – total false positive ratio as computed during the last rebuilt of the entire collection of the Bloom filters for index $idx$.

When an entire collection of filters for an index is rebuilt we allow their total size to change if this is necessary to maintain the filters’ target total false positive ratio $FP_{t_{idx}}$. Therefore, we also maintain a counter $E_{idx}$ that counts by how many bits the total actual size is over or under the budget. We will later use $E_{idx}$ to assess whether to do a complete rebuild of the index’ collection of Bloom filters. In addition, $N_{l_{idx}}$ and $N_{f_{idx}}$ are used to monitor the current false positive ratio and how much it deviates from the target ratio.
3.5.4 Triggering Filter Rebuild

The filters covering an index and each filter individually was assigned a target false positive ratio and a memory budget when built. We monitor the actual false positive ratio and trigger a rebuild when it falls outside a predefined range. When rebuilding, we size the filters for the same target ratio but allow memory usage to increase (or decrease) to achieve that target. Excessive memory usage triggers a global rebuild.

Individual Bloom Filters

The goal for individual filters is to keep the false positive ratio close to the target while keeping the actual filter size smaller than the initial.

By maintaining the access counters $N_{l_{idx,i}}$ and $N_{f_{idx,i}}$ and the two counters for new distinct keys inserted $I_{idx,i}$ and deleted $D_{idx,i}$, we can monitor the performance of a filter and take action when the performance deviates from the target. We set upper $B_u$ and lower $B_l$ bounds on the deviation from the target ratio and trigger a rebuild when the deviation exceeds the bounds. That is, a rebuild of a filter is triggered whenever

$$\frac{N_{f_{idx,i}}}{N_{l_{idx,i}}} > FP_{t_{idx,i}} + B_u$$ or $$\frac{N_{f_{idx,i}}}{N_{l_{idx,i}}} < FP_{t_{idx,i}} - B_l.$$

As mentioned earlier, access skew may cause the actual false positive ratio of a Bloom filter to be higher than what is theoretically expected. If this is the case, and we rebuild the Bloom filter with the same size, the existing keys will set the same bits as before and the bias may remain. However, if we rebuild the filter with a different size, then a different set of bits will be selected by the hash functions and, most likely, the effects of access skew will be remedied.

We can detect if enough updates have been done to disrupt the balance between set and unset bits by comparing the $I_{idx,i}$ and $D_{idx,i}$ counters with the total number of unique keys in the buckets or pages of index $idx$. If either $I_{idx,i}$ or $D_{idx,i}$ exceed a given percentage of the number of unique keys then a rebuild of the Bloom filter is triggered. In that case, the size of the filter may need to change too. The incremental algorithm described in the next paragraph achieves these restrictions.

Bloom Filters for an Index

Even if each filter in the collection retains its false positive ratio close to its target, the distribution of accesses may change causing the false positive ratio for the entire collection to deviate from its target.
In the same way as for individual filters, we can establish upper and lower bounds on the deviation from the target and trigger a rebuild of all filters for the index. To do so we need to recompute the optimal size. Since rebuilding an entire collection of filters can be costly, it should only be done if we wish to either change the overall false positive ratio or assign a new memory budget. In other cases it is better to rebuild only those Bloom filters that is needed.

3.6 Experimental Evaluation

We performed an extensive experimental study to gain insights into the behavior of Split Bloom filters. We focus on experiments that show the differences between our solution and classical Bloom filters. Since our solution is affected in the same way as a classical filter when varying the number of bits per key, or the number of hash functions, we do not study these parameters in depth. We rather focus on the effects of parameters such as the number of hash buckets per Bloom filter, or the influence of the skew of the workload, which are all specific to our approach.

We implemented our algorithms, classical Bloom filters, and an underlying storage
3.6. EXPERIMENTAL EVALUATION

The first experiment measures the creation time for Split Bloom filters and compares it with that of classical Bloom filters. Figure 3.5 plots the results. The figure shows five histogram clusters, each cluster corresponding to a different number of data keys: 1M, 5M, 10M, 20M and 40M. The bars within each cluster represent different filter
sizes: 4, 8, 16, and 32 bits per key. We plot the creation time (in milliseconds) for both classical Bloom filters and Split Bloom filters. Note that the $y$-axis is in log scale. Figure 3.5 shows that Split Bloom filters are faster to create than a large single filter. The difference in time increases with the number of keys and the number of bits per key. This is explained by the improved cache locality for Split Bloom filters. Each key added to a Bloom filter needs to set $k$ different bits. The smaller a Bloom filter is, the fewer cache lines it occupies, thus increasing the chance of a hit in the L1 or L2 cache. When the number of keys or the number of bits per key increase, the classical Bloom filters grow and the cache miss rate increases. For the same reasons of cache locality, querying and updating Split Bloom filters is faster than for a single large filter.

The number of hash buckets (or pages in a B+tree) each filter covers influences its ability to adjust to data and access skew, but also affects the total false positive. Figure 3.6 illustrates this effect. The $y$-axis of shows the overall false positive of a collection of Split Bloom filters while the exponent of the power law distribution of the workload varies between $n = 10$ until $n = 60$. The $x$-axis plots the number of hash buckets that each Bloom filter covers. In all cases, we used the same number of data keys (10 million) and the same number of total hash buckets (65,536). The solid line plots the false positive ratio of a single Bloom filter that covers all buckets and keys.
3.6. EXPERIMENTAL EVALUATION

The first observation is that Split Bloom filters are able to exploit the skew in access frequencies to reduce the false positive ratio compared to the solid line. Second, the optimal number of buckets that each Bloom filter should cover increases with the exponent of the power law distribution. For example, if $n = 10$, then 2 or 4 buckets per Bloom filter is sufficient to improve the false positive ratio. If more buckets are used per filter, then the overall false positive ratio quickly increases to that of a single Bloom filter. The reason is that the more buckets are used, the more the workload skew is “blurred out”. In the case of $n = 60$, where the access skew is high, the overall false positive ratio decreases continually until 64 buckets per Bloom filter. Therefore, the more skewed the workload is, the more buckets can be covered per Bloom filter, which in addition reduces the administration overhead.
Figure 3.7: Average cache misses and cpu cycles over 10 million queries.
3.6. EXPERIMENTAL EVALUATION

3.6.2 Queries

A disadvantage of an in memory Bloom filter is that it produces a lot of cache misses because of the random assignment of bits per key. One way to overcome this is to create hash functions that for a given key they produce values in the same cacheline range. However, this approach increases the false positive ratio since bits are not assigned truly random any more. Split Bloom filters on the other hand, are smaller by design since they cover less keys, and a smaller filter occupies less cachelines. Therefore, the probability of two or more hash values pointing at bits of the same cacheline is higher.

The next experiment demonstrates these benefits. We performed 10 million queries over a classical Bloom filter and a collection of Split Bloom filters. All queries are point lookups for keys that do exist in the filters, such that we force all hash functions to be computed each time. We report the results of our experiments in Figure 3.7. The upper row of plots show the average cache misses per key lookup. The lower row of plots show the average cpu cycles per key lookup. We repeated the experiments for different number of bits per key, from 4 bits up to 32 bits. An increase on the bits per key, also means an increase of the number of hash functions used per key. The x-axis signifies the skew of the keys stored in the buckets, according to the workloads of Figure 3.4. The goal is to demonstrate the benefits of our approach when skew increases. Figure 3.7 plots the behavior of one large Bloom filter. As expected both the cache misses and the cpu cycles remain steady per plot. However, the Split Bloom filter have a smaller number of cache misses (even for low skew) while the difference with classical Bloom filters increases for larger skewed workloads. This is also in accordance with the results of measuring the cpu cycles per key. Split filters are faster since there are fewer cache misses, but also, for less frequently accessed buckets, there are fewer hash functions to compute resulting too in reduced cpu time.

3.6.3 Access skew

Filters become less effective because of changes in access frequencies, data distribution, or inserts and deletes, and thus they need to be rebuilt. An incremental strategy, rebuilding a few filters at a time, is clearly preferable to a strategy of complete rebuilding that uses Algorithm 3.2 to compute optimal sizes. However, this raises two questions about the incremental rebuild strategy that we proposed in Section 3.5. Does it converge to a solution that is close to the optimal solution? And if it does, how rapidly does it adapt? The experiments discussed in this section are designed to shed some light on these two issues.

The first experiment investigates whether the incremental approach converges to an optimal solution. We first create a hash index with 10 million unique keys and 16,384
Bloom filters using 8 bits per key. The stored keys are uniformly distributed over the buckets. We then ran 1 million lookups selecting the target buckets according to a skewed workload with $n = 10$. During the lookups we apply the incremental rebuild strategy of Algorithm 3.3. Once the lookups complete we also compute the optimal filter sizes using Algorithm 3.2. The results are plotted in Figure 3.8. The $x$-axis enumerates the 16,384 Bloom filters, sorted from the least frequently accessed to the most frequently accessed. The $y$-axis of the upper graph shows the false positive ratio per filter and of the lower graph shows the filter’s size in bits. The plot line marked \textit{Init} shows the initial situation when access frequencies are uniform. The line \textit{IB} shows the false positive ratio and the filter size distribution after 1 million lookups and using the incremental rebuild Algorithm 3.3 to resize filters. The least accessed filters on the left of the graph are virtually never accessed so they end up with no bits and thus a false positive ratio of 1.0. However, while the access frequency of filters increase (towards the right side of the $x$-axis), more bits are allocated to the filters to keep their false positive ratio close to the target. The line \textit{RB} shows the distribution of the false positive ratio and filter size when we rebuild the entire collection of Bloom filters using Algorithm 3.2. The \textit{IB} and \textit{RB} lines are very close, which demonstrates that the incremental rebuild Algorithm 3.3 converges towards the optimal solution provided by Algorithm 3.2.

Figure 3.9 shows the evolution of the total false positive ratio and the total size of the Bloom filters when the filters are periodically rebuilt using three different strategies. A rebuild is initiated every 10,000 lookups. The $x$-axis indicates the number of lookups, up to 300,000 in total. The \textit{Init} line shows what happens if the collection of Bloom filters is rebuilt completely without changing the memory budget. The \textit{IB} line shows the results when Bloom filters are rebuilt incrementally, keeping the target false positive ratio for each filter unchanged. The \textit{RB} line corresponds to doing a complete rebuild every 10,000 lookups but by using the same total memory space as the incremental build. The lower graph of Figure 3.9 plots the total size of the filters for all three settings. The upper graph plots the corresponding overall false positive ratio.

The filters corresponding to the \textit{Init} always use the same amount of memory as shown by the straight line. Meanwhile, the overall false positive ratio drops as shown by the \textit{Init} line in the upper graph of Figure 3.9. This is caused by bits being removed from rarely accessed filters and given to more popular ones. Consequently, the total size remains the same but the overall false positive ratio is reduced and eventually converges to a minimum. The Bloom filters maintained with incremental rebuild keep the overall false positive ratio stable and close to the target, while freeing bits that are no longer needed. The overall false positive ratio is steady as shown in the upper graph of Figure 3.9, while the total size converges to a minimum as illustrated by the lower graph. The \textit{RB} line shows the results if completely rebuilding the Bloom filters
Figure 3.8: Distribution of false positive ratio and filter size across the Bloom filters of one index.
Figure 3.9: Evolution over time for a collection of Bloom filters without changing the access pattern.
after every 10,000 lookups using the same memory space as the incrementally rebuilt filter. The \textit{IB} and \textit{RB} lines are very close, which confirms that the incremental approach produces filters that are close to optimal.

The next experiment demonstrates how the incremental rebuild process adjusts to frequent changes in access patterns. The results plotted in Figure 3.9 showed that the incremental rebuild strategy responds well when there is a sudden shift in access frequencies but also that it takes time to adjust. In this experiment we simulate “moving hotspots”, that is, the skew in access frequencies remains the same but we change which buckets are popular. We vary how frequently the hotspot shifts and study the effects.

To be able to adapt to changes in access patterns, we need a mechanism to “forget history”. We have defined a simple rule for when to erase the history of access frequency. \textbf{If the total space used by the Bloom filters increases as a result of an incremental rebuild, then erase the previous history, that is, zero the counters associated with the filters.} By doing so the incremental build algorithm only takes into account recent popular Bloom filters without the picture being blurred by filters that were popular in the past.

Figure 3.10 plots the results of this experiment. The solid lines show the total size of the Bloom filters in KB while the dashed lines show the overall false positive ratio. The false positive ratio remains stable while memory consumption fluctuates because of the shifting focus. The uppermost graph of Figure 3.10 is for the case when the focus changes every 1,000 lookups. Because change is so rapid compared with the rebuild interval, memory consumption remains high, in the range 9,400 KB to 9,500 KB. In the middle graph of Figure 3.10 the focus change occurs every 10,000 lookups so the incremental build algorithm has more time to adjust. In this case, the total memory consumption varies between 9,450 KB and 8,950 KB. Finally, the lowermost graph of Figure 3.10 shows the case when the focus changes every 100,000 lookups. The incremental rebuild algorithm now has even more time to rebuild filters and reduce the total memory consumption before the focus changes again. The total memory consumption goes down to 8,700 KB.

The experiments in this section show that as long as the focus remains steady the Bloom filters will reduce their memory consumption, while still being able to adjust to more frequent shifts but at the cost of additional memory space.

### 3.6.4 Updates

As described in Section 3.5 Bloom filters may deteriorate over time due to updates. In this set of experiments we are interested in studying the effects of updates, and whether our solution can react and adjust to them. To investigate this we ran an experiment with a stream of 3 million lookups. The lookups followed a uniform access distribution, i.e.,
Figure 3.10: Incremental build every 10,000 lookups with access patterns that shift focus.
Figure 3.11: Incremental build of deteriorated Bloom filters because of updates.
all filters were equally likely to be accessed. One random insert and one random delete was done after each lookup. All lookups were performed on keys that are not present. We report the observed false positive ratio, which is computed by dividing the number of lookups that returned true (therefore are false positives) by the number of lookups processed until then.

Figure 3.11 illustrates the overall false positive ratio and the fluctuation of the total memory consumption for different rates of incremental rebuilds. We followed the following simple rule: if a Bloom filter has had more than X% of its values updated, then rebuild the filter. For example, if the limit is set to 5%, and if the total number of keys covered by a filter is 200, then after 10 updates the filter will be a candidate for rebuild. The solid line in Figure 3.11 plots the increase in the false positive ratio when no incremental rebuilds are done. The dotted line shows the total KB used by the Bloom filters when the are incrementally rebuilt because of updates. The dashed line shows the resulting observed false positive ratio. The top graph of Figure 3.11 shows the results when the update limit is at 5%. In this case the false positive ratio remains low and there are little changes in the memory space used. The middle graph of Figure 3.11 shows the case when the limit is at 10%, while the bottom graph of Figure 3.11 is for 15%. When the rebuild triggering limit is increased, incremental rebuilds occur less frequently and, as expected, causes higher fluctuation of memory space used and a worse false positive ratio.

3.7 Summary

Databases often contain both hot data and cold data. Storing cold records in memory is wasteful; it is more economical to store them on cheaper external storage. However, trips to the cold store are expensive so it is important to avoid unnecessary ones. We maintain in memory a collection of variable-size Bloom filters. Each filter covers a subset of the key values (records). This approach makes it possible to adjust the size of the filters and use more bits for subsets that are larger and/or more frequently accessed. Rebuilding one or a few small filters in response to inserts, deletes or changes in access frequencies is also faster than rebuilding a single large filter.

We derived a mathematical model and methods to optimally size the filters in a collection, taking into account the number of distinct values covered by a filter and how frequently the filter is accessed. We also showed how and when to rebuild filters in response to changes in data or access frequencies. We performed an extensive experimental evaluation and found that our approach has several advantages compared with using a single large filter. They are faster to build; they take advantage of skew to achieve a lower false positive rate; and they can adapt to changes in both data and
3.8 Appendix A. Optimal Sizing of Split Bloom Filters

Consider a Bloom filter for the $i$-th group of hash buckets or B-tree pages that has $m_i$ bits and uses $k_i$ hash functions. The expected proportion of bits still set to 0 after $n_i$ records have been added equals

$$\varphi_i = (1 - \frac{1}{m_i})^{k_i n_i}$$

(3.8)

For large enough $n_i$ the following approximation holds

$$\varphi_i = (1 - \frac{1}{m_i})^{k_i n_i} \approx e^{-k_i n_i / m_i}$$

(3.9)

It follows that the expected fraction of false positives for group $i$ is

$$p_i = (1 - \varphi_i)^{k_i} \approx (1 - e^{-k_i n_i / m_i})^{k_i}$$

(3.10)

It can be shown that Equation 3.10 is minimized when $k_i = \ln 2 / m_i$. Therefore, if $m_i$ and $n_i$ are given, it is possible to choose an optimal value for $k_i$. Using the optimal $k_i$ value we have

$$p_i \approx 2^{-\ln 2} \frac{m_i}{m}$$

(3.11)

Given $i)$ $l$ groups, $ii)$ the number $n_i$ of (distinct) records of each group such that $\sum_{i=1}^{l} n_i = n$ is the total number of records, and $iii)$ the number $f_i$ of access frequencies of a group such that $\sum_{i=1}^{l} f_i = f$, we wish to distribute $m$ total bits over the $l$ groups such that $\sum_{i=1}^{l} m_i = m$ and the total expected fraction of false positives of the corresponding $l$ Bloom filters is minimized. The total expected false positives $P$ is given by the following Equation

$$P = \sum_{i=1}^{l} \frac{f_i}{f} p_i$$

(3.12)

We solve the optimization problem of minimizing the sum of Equation 3.12 using Lagrange multipliers. The Lagrange function is defined as follows.

$$\Lambda(\vec{m}, \lambda) = P(\vec{m}) + \lambda(g(\vec{m}) - m)$$

(3.13)
where \( \vec{m} = m_1, \ldots, m_l \), and

\[
P(\vec{m}) = \sum_{i=1}^{l} \frac{f_i}{f} 2^{-\ln 2 \frac{m_i}{n_i}}
\]

(3.14)

and

\[
g(\vec{m}) = \sum_{i=1}^{l} m_i
\]

(3.15)

We wish to solve the following system

\[
\nabla_{\vec{m}, \lambda} \Lambda(\vec{m}, \lambda) = 0
\]

(3.16)

The partial derivative \( \frac{\partial \Lambda}{\partial m_i} \) for all \( m_i, i = 1, \ldots, l \) is

\[
\frac{\partial \Lambda}{\partial m_i} = \frac{f_i}{f} \left( -2^{-\ln 2 \frac{m_i}{n_i}} \ln 2 n_i \right) + \lambda
\]

(3.17)

and for \( \lambda \)

\[
\frac{\partial \Lambda}{\partial \lambda} = m_1 + m_2 + \ldots + m_l - m
\]

(3.18)

Substituting Equation 3.17 into 3.16

\[
\frac{f_i}{f} \left( -2^{-\ln 2 \frac{m_i}{n_i}} \ln 2 n_i \right) + \lambda = 0
\]

(3.19)

\[
m_i = -n_i \ln \left( \frac{f \lambda n_i}{f_i \ln^2 2} \right)
\]

(3.20)

similarly, substituting Equation 3.18

\[
\sum_{i=1}^{l} m_i - m = m_1 + m_2 + \ldots + m_l - m = 0
\]

(3.21)
Next, we combine Equation 3.20 and 3.21 to solve for $\lambda$

$$\sum_{i=1}^{l} n_i \ln \left( \frac{f_i \ln^2 2}{f \lambda n_i} \right) = m$$

(3.22)

$$\sum_{i=1}^{l} \left( n_i \left( \ln ( f_i \ln^2 2) - \ln (f \lambda n_i) \right) \right) = m \ln^2 2$$

(3.23)

$$\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i} + \left( \ln \ln^2 2 - \ln f - \ln \lambda \right) \sum_{i=1}^{l} n_i = m \ln^2 2$$

(3.24)

$$\ln \lambda = \frac{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i}}{n} + \ln \ln^2 2 - \ln f - \frac{m}{n} \ln^2 2$$

(3.25)

$$\lambda = e^{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i} / n} e^{-\frac{m}{n} \ln^2 2 \ln^2 2 f^{-1}}$$

(3.26)

Substituting $\lambda$ from Equation 3.26 into 3.20 gives

$$m_i = \frac{n_i \ln \frac{f_i}{n_i} \ln^2 2}{f e^{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i} / n} e^{-\frac{m}{n} \ln^2 2 \ln^2 2 f^{-1}} \ln^2 2}$$

(3.27)

Therefore

$$m_i = \frac{n_i \ln \frac{f_i}{n_i}}{\ln^2 2}$$

(3.28)

where

$$\varpi = e^{-\frac{m}{n} \ln^2 2} e^{\sum_{i=1}^{l} n_i \ln \frac{f_i}{n_i} / n}$$

(3.29)

To verify the correctness of our calculations notice that, if $f_i = f_j$ and $n_i = n_j$, for all $i, j \in 1, \ldots, l$, then

$$\varpi = e^{-\frac{m}{n} \ln^2 2} e^{\frac{m}{l} \ln \frac{f_i}{n_i}} = e^{-\frac{m}{n} \ln^2 2 \frac{f_i}{n_i}}$$

(3.30)

and

$$m_i = n_i \ln e^{\frac{m}{n} \ln^2 2 \ln^2 2} = \frac{n_i m}{n} = \frac{m}{l}$$

(3.31)

which yields the same number of bits per group, since all groups have the same frequency and the same number of (distinct) records. Similarly, if we assume that $f_k =$
which correctly yields a solution where all bits are assigned to the group that contains all the records and gets all the queries.

In order to ease the numerical calculation of \( m_i \), we rewrite Equation 3.28 as

\[
m_i = (\ln^{-2} 2\gamma + \frac{m}{n})n_i \tag{3.34}
\]

where

\[
\gamma = \ln f_i - \ln n_i + \sigma \tag{3.35}
\]

\[
\sigma = \frac{1}{n} \sum_{i=1}^{l} n_i (\ln n_i - \ln f_i) \tag{3.36}
\]

while \( \sigma \) is constant for all pages.

Moreover, in order for 3.34 to give correct results, that is not negative values, the following must hold:

\[
\ln^{-2} 2\gamma + \frac{m}{n} > 0 \tag{3.37}
\]

\[
\gamma > -\frac{m}{n} \ln^2 2 \tag{3.38}
\]

\[
\ln n_i - \ln f_i < \sigma + \frac{m}{n} \ln^2 2 \tag{3.39}
\]

which holds if \( f_i \ll f \) or if \( f_i \) is reasonably close to or greater than \( n_i \). If \( f_i \) is very small then we assign \( m_i = 0 \) and assume that no Bloom filter is needed since these group of hash buckets or B+tree pages are rarely if ever queried.

3.9 Appendix B. Incremental Sizing of Split Bloom Filters

A Bloom filter that covers the \( i \)-th group of hash buckets or B+tree nodes contributes to the overall false positive ratio by \( \frac{f_i}{f} p_i \) where \( p_i \simeq 2^{-\ln^2 \frac{m_i}{n_i}} \). Assume that the access
frequency of this Bloom filter has changed to \( f'_i \), or the number of unique records to \( n'_i \), or both. We need to readjust \( m_i \) to a new value \( m'_i \) such that the new contribution to the overall false positive remains the same. More formally, the following equality should hold

\[
\frac{f_i}{f} p_i = \frac{f'_i}{f'} p'_i
\]

(3.40)

where now

\[
p'_i \approx 2^{-\ln \frac{m'_i}{m_i}}
\]

(3.41)

and

\[
p'_i = \frac{f'_i}{f'_i} \frac{f_i}{f} p_i
\]

(3.42)

We solve Equations 3.41 and 3.42 for \( m'_i \)

\[
m'_i = -n'_i \ln \left( \frac{f'_i}{f'_i} \frac{f_i}{f} p_i \right) \ln^{-2} 2
\]

(3.43)

By keeping for each Bloom filter the targeted false positive ratio \( p_i \) and the fraction \( \frac{f_i}{f} \) we adjust the filter size to \( m'_i \) bits in order to maintain the same contribution to the overall false positive.

It is possible that \( f'_i \) becomes so small that \( p'_i > 1 \). In this case, we set \( p'_i = 1 \) and consequently \( m = 0 \). Note that if \( m = 0 \), then the Bloom filter always returns true.