Distributed and Parallel Databases manuscript No.
(will be inserted by the editor)

Cardinality estimation and dynamic length adaptation for Bloom filters

Odysseas Papapetrou · Wolf Siberski · Wolfgang Nejdl

Abstract Bloom filters are extensively used in distributed applications, especially in distributed databases and distributed information systems, to reduce network requirements and to increase performance. In this work, we propose two novel Bloom filter features that are important for distributed databases and information systems. First, we present a new approach to encode a Bloom filter such that its length can be adapted to the cardinality of the set it represents, with negligible overhead with respect to computation and false positive probability. The proposed encoding allows for significant network savings in distributed databases, as it enables the participating nodes to optimize the length of each Bloom filter before sending it over the network, for example, when executing Bloom joins. Second, we show how to estimate the number of distinct elements in a Bloom filter, for situations where the represented set is not materialized. These situations frequently arise in distributed databases, where estimating the cardinality of the represented sets is necessary for constructing an efficient query plan. The estimation is highly accurate and comes with tight probabilistic bounds. For both features we provide a thorough probabilistic analysis and extensive experimental evaluation which confirm the effectiveness of our approaches.

Note: This is a preprint. The final version is available at `http://www.springerlink.com/`

1 Introduction & Motivation

Bloom filters were proposed in [1] as compact approximate set representations. The standard application of Bloom filters is for representing sets in a format suitable for answering membership queries, i.e., whether an element x is member of a set S. Bloom filters enable

Odysseas Papapetrou
L3S Research Center, Leibniz Universität Hannover
E-mail: papapetrou@l3s.de

Wolf Siberski
L3S Research Center, Leibniz Universität Hannover
E-mail: siberski@l3s.de

Wolfgang Nejdl
L3S Research Center, Leibniz Universität Hannover
E-mail: nejdl@l3s.de

answering membership queries in constant time, and with a configurable false positive probability. They improve upon alternative representations with respect to required memory and construction time. In particular, Bloom filter creation cost and memory requirements are linear with set size, with a very low constant.

Since their proposal, Bloom filters have found wide use in distributed databases, for reducing network costs. For instance, the Bloom join algorithm [2–4], and several extensions [5, 6] reduce substantially the network cost of distributed joins, by representing the join attributes with Bloom filters. KLEE [7, 8] uses Bloom filters to optimize the network usage in distributed top-k query execution. Peer-to-peer applications use Bloom filters to represent peer contents, to enable query routing in unstructured P2P networks [9–12], and for estimating item novelty [13]. In addition, Bloom filters are used for optimizing collaboration protocols, such as collaborative caching [14] and content reconciliation [15], as well as for query optimization in databases with confidential data, to enable join execution without revealing information [16]. In general, the compactness of Bloom filter representations and the constant cost for membership tests is appealing for a wide range of data-intensive distributed systems.

In line with their significance, Bloom filter characteristics have been analyzed in depth, and several extensions have been proposed, e.g., [17–20]. In the context of our work on distributed databases and peer-to-peer systems we devised two additional features: (a) the ability to dynamically change the Bloom filter length to make it more appropriate for particular requirements, e.g., the number of elements it represents, and, (b) the ability to estimate the number of distinct elements hashed in a Bloom filter. These features are important for many distributed settings. The first feature, dynamically adapting the Bloom filter length, is important for optimizing network usage in distributed databases, e.g., in Bloom joins. We implement this feature with an extension on the Bloom filter structure, called *Block-partitioned Bloom filters*. The second feature, deriving a cardinality estimation of a set from its Bloom filter representation, is required for counting the distinct elements in non-materialized sets, e.g., streams, or partial joins with the Bloom join algorithm. We propose a probabilistic *Cardinality Estimation* approach, which estimates the number of elements in a Bloom filter based on its density, efficiently and with high accuracy. A preliminary, limited version of this work was presented in [5]. Source code for both features is available online[1].

Block-partitioned Bloom filters. We first examine how to dynamically reduce the length of a Bloom filter. Standard Bloom filters are bound to the length decided during their initialization. However, in many cases, it is desirable to reduce the Bloom filter length dynamically, that is, after all elements have been inserted. For example, when executing a chain of Bloom joins in distributed databases [5], the intermediary Bloom filters may become too sparse, wasting network resources. In addition, for some applications, the optimal Bloom filter length cannot be computed a priori. For instance, for the case of Bloom joins, the optimal Bloom filter length depends on parameters determined for each particular Bloom join at runtime [6]. Similarly, when Bloom filters are used for summarizing large sets in distributed systems, e.g., [14, 21], the optimal Bloom filter length for each communication is determined from properties of each particular communication, such as the network connectivity between the two endpoints.

Currently, the only way to reduce the Bloom filter length is by rebuilding it from scratch, which is computationally very expensive. When non-materialized sets are involved, e.g., elements of a stream that are not saved locally, rebuilding the filter from scratch is even infeasible. To address this problem, we propose a novel encoding scheme for Bloom filters. We

[1] `http://www.l3s.de/~papapetrou/ebf/ebf.jar`

call it *Block-partitioned Bloom filter* because it partitions the Bloom filters in smaller blocks, where each subset of blocks can act as an independent Bloom filter. Reducing the length of a Block-partitioned Bloom filter incurs practically no cost, and causes only a negligible loss of accuracy compared to the optimal Bloom filter of the same length.

An established extension of Bloom filters is Dynamic Bloom filters [17, 22], which enables the filter to increase its length for accomodating more elements. We combine Block-partitioned Bloom filters with Dynamic Bloom filters, to enable both increasing and reducing the length, allowing for a flexible adaptation to the contents with a near-optimal false positive probability. The combination is called ***Dynamic Block-partitioned Bloom filters***, and it is useful for summarizing sets that increase progressively, e.g., [15, 23]. Standard Bloom filters are inefficient for these contexts, as they need to be initialized for the worst-case scenario, i.e., for a very large cardinality. Dynamic Block-partitioned Bloom filters enable a pay-as-you-go approach, starting from a generic Bloom filter length, and increasing or reducing it progressively.

Cardinality Estimation for Bloom filters. The second contribution allows estimating the Bloom filter cardinality in the absence of the represented set[2]. At first glance, it looks as if this problem could be easily solved by attaching the set cardinality to the Bloom filter. However, this is not always possible, because Bloom filters are often used to represent non-materialized sets. For example, the Bloom filter of the union (intersection) of two sets can be computed efficiently by performing a bitwise disjunction (conjunction) of their respective Bloom filters, without actually requiring the materialization of the resulting set or any of the two sets. This case frequently occurs in distributed databases, e.g., Bloom joins [2–4], where estimating the cardinalities is used to create an optimal query plan [6]. Furthermore, in P2P networks, estimating the Bloom filter cardinality is required for devising query plans that increase the information retrieval quality, e.g., [13], as well as for organizing the overlay network such that peers with similar contents become neighbors [24]. We present an in-depth analysis of probabilistic cardinality estimation, and derive tight probabilistic error bounds. Our analysis also covers Block-partitioned Bloom filters, and Bloom filters generated by bitwise conjunction or disjunction of standard Bloom filters. Extensive experimental evaluation shows that the estimations are highly accurate, even for extremely dense Bloom filters.

In the rest of the paper, we describe, discuss, and evaluate each contribution in depth independently, and also show how they are combined. Since the proposed contributions are applicable to a variety of contexts, we deliberately do not constrain the experimental evaluations to a particular application scenario. Instead, we cover a broad range of applications and contexts through extensive experimentation. The paper is structured as follows. In the next section we present the basics for Bloom filters, and introduce notations. In Section 3 we describe Block-partitioned Bloom filters, and in Section 4 we extend them to Dynamic Block-partitioned Bloom filters. The probabilistic cardinality estimation approach is presented in Section 5, for both standard and Block-partitioned Bloom filters. Section 6 presents further usage scenarios and applications for the three proposed extensions, mostly from the area of distributed databases and information systems. We close with related work and conclusions.

[2] We use the expression 'Bloom filter cardinality' to denote the cardinality of the set represented by this Bloom filter

2 Bloom filter basics

A Bloom filter is a space-efficient representation of a set $S = \{x_1, x_2, x_3, \ldots, x_n\}$ of n elements from a universe $\mathcal{U}$. It consists of an array of m bits and a family of k independent hash functions $F = \{f_1, f_2, \ldots, f_k\}$, which hash elements of $\mathcal{U}$ to integers in the range of $[1, m]$. All m bits are set to 0 initially[3]. An element x is inserted into the Bloom filter by setting all positions $f_i(x)$ of the bit array to 1.

We assume that an element x is contained in the original set if all positions $f_i(x)$ of the Bloom filter are equal to 1. If at least one of these positions is set to 0, then we conclude that x is not present in the original set. However, Bloom filters exhibit a small probability of false positives; due to hash collisions, it is possible that all bits representing a certain element have been set to 1 by the insertion of other elements. The probability for a false positive is $Pr_{fp} \approx (1 - e^{-kn/m})^k$.

In some scenarios, the number of elements hashed in a Bloom filter is unknown. For these cases, we can compute the false positive probability based on the number of true bits in the filter. A false positive occurs when all k hash values point to true bits. For a Bloom filter with t bits set to true, the probability that one hash function points to a true bit equals to t/m. The probability that all k hash functions point to true bits, which leads to a false positive, is $(t/m)^k$.

For given set cardinality and Bloom filter length, the false positive probability can be minimized by optimizing the ratio between true bits and Bloom filter length. We denote this ratio as *Bloom filter density*. The false positive probability is minimized when this density is 0.5. This is the case when the number of hash functions is set to $k \approx \frac{m}{n} \ln(2)$.

Bloom Filter resolution. Clearly, the false positive probability is influenced by the length of the Bloom filter m and the number of hash functions k. We refer to a Bloom filter configuration consisting of these two parameters as its resolution, because it conveys the ability of the filter to represent a set.

2.1 Set Union and Intersection with Bloom Filters

It is often convenient to perform approximate set union and intersection directly on the Bloom filters of the sets. For example, in distributed settings nodes can perform intersection of their respective Bloom filter representations to identify overlapping content. We now present these operations and the accompanied false positive probabilities for each operation.

Set union with Bloom filters. We can construct the Bloom filter corresponding to the union of two sets S_1 and S_2 by merging bitwise their Bloom filters BF_1 and BF_2. In particular, the merged Bloom filter $BF_\cup$ of set $S_\cup = S_1 \cup S_2$ equals to $BF_1 \vee BF_2$, with $\vee$ denoting a bitwise OR merging.

Since the Bloom filter $BF_\cup$ constructed with OR-merging is identical to the traditional Bloom filter of $S_\cup$, its false positive probability can be found as explained previously for standard Bloom filters; if the number of elements $n_\cup$ in the union is known, the false positive probability in $BF_\cup$ is $Pr_{fp}[BF_\cup] \approx (1 - e^{-kn_\cup/m})^k$, where m and k denote the length of the filter and the number of hash functions respectively. If $n_\cup$ is unknown, for instance, when the intersection is not materialized, the probability for a false positive can be inferred from the number of true bits t in the Bloom filter: $Pr_{fp}[BF_\cup] = (t/m)^k$.

[3] We use the expressions ‘A bit is set to true/false’ and ‘A bit is set to 1/0’ interchangeable.

m	Length of Bloom filter	$\hat{S}^{-1}(\cdot)$	Maximum likelihood value of the number of elements
k	Number of hash functions		
n	Number of elements in the Bloom filter	μ	Number of blocks for BBFs and D-BBFs
Pr_{fp}	False positive probability		
c	Expected set cardinality	λ	Number of batches for D-BBFs
t	Number of true bits in the Bloom filter	n_{thres}	Max. number of elements per batch in D-BBFs
$\hat{S}(\cdot)$	Maximum likelihood value of the number of true bits		

Table 1 Notations used throughout Sections 3 to 5

Set intersection with Bloom filters. Let $BF_\cap$ denote the Bloom filter of $S_1 \cap S_2$. The Bloom filter representations of S_1 and S_2 are not sufficient for accurately computing $BF_\cap$. We can however get an approximation by joining BF_1 and BF_2 with a bitwise-AND.

Let $BF_\wedge := BF_1 \wedge BF_2$, with $\wedge$ denoting bitwise AND. Then $BF_\cap \approx BF_\wedge$. Although $BF_\wedge$ is an approximation, it can still be used for membership tests in the same way as standard Bloom filters: we conclude that an element x is not contained in $S_1 \cap S_2$ if at least one of the hash values of x points to a false bit in $BF_\wedge$. If all the functions for x map to true bits in $BF_\wedge$, with high probability x belongs to $S_1 \cap S_2$.

Similar to the case of standard Bloom filters, a false positive occurs when all k hash values of an element point to true bits in $BF_\wedge$. Each hash value points to a true bit with probability t/m, where t denotes the number of true bits in $BF_\wedge$, and m its length. The probability that all hash values point to true bits is $Pr_{fp}[BF_\wedge] = (t/m)^k$, where k denotes the number of hash functions. Since t is less than or equal to the true bits in any of the Bloom filters BF_1 and BF_2, the following inequalities also hold: $Pr_{fp}[BF_\wedge] \leq Pr_{fp}[BF_1]$, and $Pr_{fp}[BF_\wedge] \leq Pr_{fp}[BF_2]$).

An interesting observation is that the cardinality of the intersection is not easily computable since the actual set intersection is not materialized. The same applies to set union. In Section 5.2 we show how to estimate this cardinality with high precision, using only the Bloom filters.

3 Block-Partitioned Bloom Filters for Resolution Reduction

We now consider the problem of adapting the Bloom filter resolution dynamically. In particular, we want to reduce the Bloom filter resolution according to application requirements, after the Bloom filter has been constructed. This is required for optimizing Bloom filters when they cannot be recreated from scratch, e.g., for streaming data. But even in cases where recreating a Bloom filter of optimal length would be possible, we would like to avoid the computationally expensive rehashing of all elements. Therefore, we require a technique which enables us to reduce the resolution of a Bloom filter without rehashing the elements, and even in the absence of the set that the Bloom filter represents.

In the next section we point out why standard Bloom filters are inapt for this purpose. In Section 3.2 we describe and analyze Block-partitioned Bloom filters, which efficiently address the problem of resolution reduction. We validate their efficiency and effectiveness, both theoretically, and experimentally in Section 3.3. Table 1 summarizes the definitions used throughout the rest of this paper.

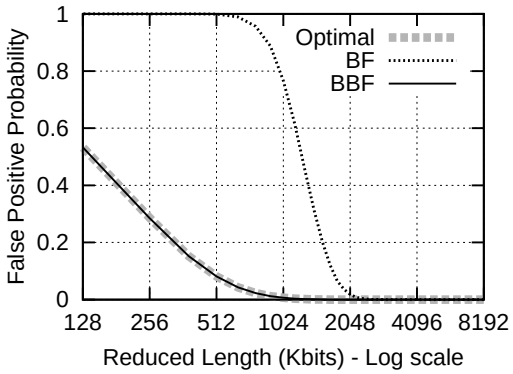

Fig. 1 False positive probabilities with different Bloom filter reduction techniques.

3.1 Resolution Reduction for Standard Bloom Filters

Assume that we want to reduce a large sparse Bloom filter BF of length m to a smaller one of length m', denoted with BF'. For this, we need a transformation function $map(\cdot)$ which maps the bits from the original filter to the target Bloom filter, e.g., $map(x) = (x \bmod m')$. For testing whether an element belongs to the reduced Bloom filter BF', we need to use the original hash functions, then apply map, and finally check if the respective bits are set in BF'. The crucial issue here is that, regardless of the choice of the mapping function, we cannot reduce the number of hash functions in this process, because for any given bit in BF, it is impossible to find out which hash function(s) set it to true.

Given the length ratio $r = m'/m$, the false positive probability for BF' is $Pr'_{fp} \approx (1 - e^{-kn/(mr)})^k$. The false positive probability before reduction is $Pr_{fp} \approx (1 - e^{-kn/m})^k$, while the optimal false positive probability for a Bloom filter of length m' which represents the same set is only $Pr_{\text{reduced-opt}} \approx (1 - e^{-k_{opt}n/(mr)})^{k_{opt}} = (1 - e^{-knr/(mr)})^{kr} = Pr_{fp}^r$. It follows that $Pr_{fp} < Pr_{\text{reduced-opt}} < Pr'_{fp}$ for any length ratio $0 < r < 1$. This result is independent of the chosen transformation function.

Figure 1 shows how the Bloom filter false positive probability increases when a standard Bloom filter is reduced using mapping, compared to the optimal reduction. The probability for the optimal reduction is computed by rebuilding the Bloom filter from scratch, using the optimal number of hash functions for the given number of elements and Bloom filter length. The results are for an initial Bloom filter of length 8192 Kbits, representing 100000 elements. The initial Bloom filter uses 60 hash functions, which is the optimal number for this configuration. For the standard Bloom filter, false positive probability already exceeds 0.75 when the Bloom filter length is reduced to 1/8 of the original length using mapping. Note that the optimal false positive probability for a Bloom filter of the same length is less than 0.01.

The reason for the large increase in the false positive probability is the inability to reduce the number of hash functions in proportion to the length of the Bloom filter. Therefore, the density of the reduced Bloom filter increases to values higher than the optimal density, which is 0.5 [25]. The false positive probability increases polynomially with the density. Consequently, the Bloom filter soon becomes unusable for membership tests.

3.2 Block-partitioned Bloom Filters

The key for maintaining a near-optimal false positive probability while reducing the Bloom filter length is to ensure that its density remains around 0.5. To achieve this, we need to adapt the number of hash functions to the reduced length.

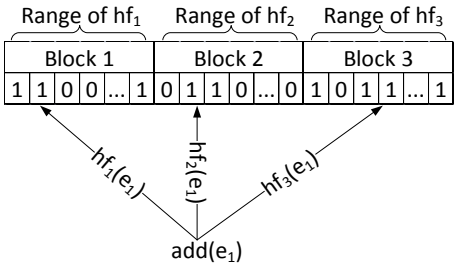

Fig. 2 Adding an element to a Block-partitioned Bloom filter of 3 blocks and 1 hash function per block.

To address this requirement, we propose Block-partitioned Bloom filters (**BBFs**). BBFs are composed of many small, independent Bloom filters, each with its own bit array and hash functions. We refer to these smaller Bloom filters as blocks. In particular, let m be the desired length of the BBF. We compose the BBF by concatenating μ blocks, each of length $m_b = m/\mu$ and with its own k_b hash functions. Similar to standard Bloom filters, all hash functions in a BBF are pairwise independent. For the purpose of adding elements and checking for elements, each block is treated as a stand-alone Bloom filter. An element is added to the BBF by adding it to all blocks (cf. Figure 2). Membership tests are also performed against all blocks. If the membership test fails for any block, then the element is not contained in the BBF. A false positive occurs when all blocks answer positively for an element because of hash collisions, even though the element had not been hashed in the BBF.

Assuming pairwise independent hash functions in all blocks, we find the false positive probability of the BBF by multiplying the corresponding probabilities for each block. If n elements are already hashed in the BBF, the false positive probability in each block is $Pr_{fp\text{-}block} = (1 - (1 - \frac{1}{m_b})^{k_b n})^{k_b} \approx (1 - e^{-k_b n/m_b})^{k_b}$, where k_b denotes the number of hash functions per block and m_b the length of each block. This gives us a total false positive probability $Pr_{fp} = \left(1 - \left(1 - \frac{1}{m_b}\right)^{k_b n}\right)^{k_b m/m_b} \approx (1 - e^{-k_b n/m_b})^{k_b m/m_b}$.

Reducing the length of a BBF is straightforward and incurs practically no cost. Consider a BBF with μ blocks, each of length m_b. We reduce it to a length $m' \leq \mu \times m_b$ by taking only the first m'/m_b blocks (rounded to the nearest integer) with their accompanying hash functions. The resulting Bloom filter can also undergo the same process again, if further resolution reduction is required. This reduction step is inexpensive regarding memory and computation, since it does not require rehashing of the elements. Therefore BBFs are also suitable for use in applications with real-time constraints, such as stream summarization.

It is important to note that BBFs can still be combined with bitwise AND, or bitwise OR, similar to normal Bloom filters. Even more important for networking applications, we can also combine BBFs of different resolutions as long as these share the same hash functions and block length; in this case, we produce a combined BBF of the lowest of the two resolutions. Particularly concerning the bitwise OR case, more advanced merging techniques are

also possible, such as keeping the additional information from the largest BBF separately, and adapting the membership tests accordingly. However, such merging techniques have the disadvantages that they do not yield standard BBFs, they require keeping track of all the merging actions, and inevitably add complexity at the membership test algorithm.

Configuration of Block-partitioned Bloom Filters. We now show how a BBF is initialized to minimize the false positive probability. Let m denote the maximum length that the BBF can occupy. The expected number of elements to be hashed in the BBF is denoted with n. For configuring the BBF, the optimal length per block m_b, number of hash functions per block k_b, and number of blocks μ need to be chosen. We achieve maximum length flexibility by setting k_b to 1. With respect to μ and m_b, we want the configuration that minimizes the false positive probability for the BBF, subject to $m_b \times \mu = m$. The configuration $m_b = \lceil n/\ln(2) \rceil$ and $\mu = m/m_b$ is the one that results to an expected density of 50% in each block (the optimal information theoretic density), and thus minimizes the false positive probability. The resulting false positive probability for the BBF then becomes:

$$Pr_{fp} = \left(1 - \left(1 - \frac{1}{m_b}\right)^{n}\right)^{m/m_b} \approx \left(1 - e^{-n/m_b}\right)^{m/m_b} \quad (1)$$

In some scenarios, m_b is preselected from the application requirements, e.g., so that each block can nicely fit to the processor's L1/L2 cache or for enabling the BBF to be reduced to specific lengths. Then, we find the optimal value for k_b as follows. False positive probability of the BBF is minimized when the false positive probability for each block is minimized. Given the expected number of hashed elements n, the number of hash functions per block k_b that minimize this probability is $k_b = max\,(1, \lfloor (m_b/n)\ln(2) \rfloor)$.

3.3 Evaluation

We evaluated the false positive probability of BBFs experimentally. In particular, we compared the false positive probability achieved by BBFs with the respective probability achieved when reducing a standard Bloom filter, using mod as a mapping function, as explained in Section 3.1. As an additional baseline, we have used the false positive probability exhibited by the optimal Bloom filter of the same length, i.e., there exists no other Bloom filter configuration which achieves a lower false positive probability for this length. To compute this baseline, we determined the optimal number of hash functions for the given length and set cardinality, and rebuilt the standard Bloom filter from scratch.

Our evaluation setup simulates the scenario of reducing the length of a Bloom filter before sending it over the network, as is frequently required in distributed applications (see Section 6). We ran the same set of experiments for sets with cardinalities between 50000 and 1 million. We now present the results for a set size of 100000. The outcomes for the other cardinalities were similar. We generated the set by selecting 100000 distinct elements randomly. Due to hashing, the presented evaluation results are independent of the type of elements contained in the set, which in our case were randomly selected integers. After constructing the set, we hashed all its elements in a standard Bloom filter of 8192 Kbits with 60 hash functions, which minimized the false positive probability. For the BBF, we used 64 blocks of 128 Kbits, each with 1 hash function.

Figure 1 shows the effect of length reduction on the false positive probability (X-axis is log scale). We see that BBFs exhibit near-optimal false positive probability for all reduction lengths. On the contrary, the standard Bloom filter suffers from high false positive probabilities even for relatively small reductions. For instance, when the standard Bloom filter is

reduced to 1024 Kbits, the false positive probability exceeds 0.75, while the false positive probability given by the BBF for the same length is less than 0.01. Reducing the length further to 512 Kbits renders the standard Bloom filter useless for all practical concerns, while the corresponding optimal and Block-partitioned Bloom filter still maintain false positive probabilities less than 0.1, which are still acceptable for a wide range of network applications, e.g., [6, 11, 21, 26]. We also note that, for all reduction lengths, the false positive probability offered by the BBF is nearly equal to the optimal false positive probability: the difference between the corresponding false positive probabilities in the above example is always less than 0.0001.

False positive probability of standard Bloom filters increases drastically with resolution reduction because their density becomes too high, as a result of the fixed number of hash functions. For example, when the standard Bloom filter is reduced to 1024 Kbits, its density exceeds 0.99, and its false positive probability is 0.76. The optimal false positive probability for a standard Bloom filter of this length is obtained with 7 hash functions and it is only 0.007. Approximately the same false positive probability is obtained by the BBF which uses 8 blocks, each with one hash function. The density of each block in the BBF is independent of the total number of hash functions, therefore the false positive probability increases slowly compared to standard Bloom filter resolution reduction.

In summary, Block-partitioned Bloom filters enable efficient and effective resolution reduction, and can be applied even in applications with real-time requirements. They exhibit near-optimal false positive probability for all reduction rates, without requiring rehashing of elements. Moreover, they enable all basic Bloom filter operations, like membership queries, and Bloom filter unions and intersections.

4 Dynamic Block-partitioned Bloom Filters

Applications frequently require Bloom filters that can be both reduced and increased in length. Consider for instance applications which construct Bloom filters of sets of unknown cardinalities, e.g., non-materialized sets, or streams. These applications cannot optimize a priori the Bloom filter configuration for the cardinality of the set. If they underestimate the set cardinality, the false positive probability of the Bloom filter will be too high and will render the Bloom filter useless. If they overestimate the set cardinality, they will generate a very large Bloom filter which cannot be easily sent over the network or stored. These scenarios require a pay-as-you-go solution, a data structure that inherits the Bloom filter characteristics and that can be increased and/or reduced in length to adjust to the set cardinality, or to the requirements of the application.

Block-partitioned Bloom filters go a step towards the right direction; they allow reducing the Bloom filter length and number of hash functions at will, so that the application-specific cost function is optimized. However, the structure itself does not allow increasing the Bloom filter length, to compensate for a higher number of elements than the expected one. We address this limitation by combining BBFs with an orthogonal approach, the Dynamic Bloom filters proposed by Guo et al. in [17, 22].

The Dynamic Bloom filters approach proposes starting with a single small Bloom filter, and continuing to add elements in it until the number of hashed elements reaches a predefined threshold n_{thres}. Then, a new empty Bloom filter is constructed, with the same hash functions and the same length, and is attached to the data structure. The next n_{thres} elements are then added to the new Bloom filter. This process is repeated until all set elements are hashed. Query processing for the Dynamic Bloom filter is analogous; for finding whether an

	Range of hf_1						Range of hf_2						Range of hf_3					
Batches	Block 1						Block 2						Block 3					
Batch 1	1	1	1	0	...	0	1	1	1	0	...	0	0	1	0	1	...	1
Batch 2	0	1	0	1	...	1	0	1	1	0	...	0	0	1	1	0	...	0
Batch 3	0	1	1	0	...	0	1	0	1	0	...	1	0	1	0	0	...	1
Batch 4	0	0	0	0	...	0	0	0	0	0	...	0	0	0	0	0	...	0

Fig. 3 A Dynamic Block-partitioned Bloom filter with 4 batches and 3 blocks per batch. The first three batches are frozen – no additions are allowed. New elements are added to the last batch.

element exists in a Dynamic Bloom filter, the element is hashed once, and checked against all Bloom filters of the data structure. If the element does not exist in any of them, we can safely conclude that the element is not in the set. If on the other hand one of the filters returns a positive answer for an element, the element exists in the original set with a computable probability. For a Dynamic Bloom Filter with n elements, the false positive probability is at most equal to $1-\left(1-\left(1-e^{-kn_{thres}/m}\right)^{k}\right)^{\lceil n/n_{thres}\rceil}$, where m and k denote the length and number of hash functions in each Bloom filter respectively.

The false positive probability of Dynamic Bloom Filters grows almost linearly with the number of elements. Therefore, when the number of elements can be approximated, normal Bloom filters are better suited for representing the set. Similarly, when the set cardinality can be upper bounded, a BBF can be used, which can be reduced after the whole set has been hashed, to achieve the required trade-off between length and false positive probability. However, when the set cardinality cannot be approximated at all, Dynamic Bloom Filters are the only viable option.

We combine BBFs with Dynamic Bloom Filters to get Bloom filters of a fully adjustable length. We refer to the new structure as Dynamic Block-partitioned Bloom filters (**D-BBF** for short). D-BBFs offer the necessary functionalities for reducing and increasing the Bloom filter length, to dynamically adapt to the cardinality of the set. The structure works as follows (for now, assume that the optimal configuration parameters for the D-BBF are given). Blocks are considered in batches, as depicted in Figure 3. For blocks of length m_b and with k_b hash functions, we set the threshold of maximum elements per batch n_{thres}. We then initialize the first batch of blocks, which is essentially a Block-partitioned Bloom Filter, and we start hashing the elements. When a batch reaches its maximum elements threshold, we freeze all blocks in the current batch, and create a new batch of blocks for hashing the rest of the elements. The process is repeated until all elements are hashed. Querying for an element follows the same logic: all Bloom filter blocks on all batches are independently queried. If a batch of blocks is found which appears to contain the query, then the element belongs at the original set with high probability. If no batch of blocks fully satisfies the query, then the element does not belong in the set represented by the D-BBF. To reduce a D-BBF, each of the contained BBFs is reduced to the new length, as explained in Section 3. Optimally, the reduction process occurs after all elements are hashed in the D-BBF, such that the precise false positive probability, can be determined or upper-bounded. The reduced D-BBF remains completely functional, i.e., it can accept more elements, answer membership queries, and undergo further reduction.

We now compute the false positive probability for the D-BBF. Let n denote the total number of elements hashed in the D-BBF. With n_{thres} we represent the threshold of maximum elements per batch. We use λ to denote the total number of batches, i.e., $\lambda = \lceil n/n_{thres} \rceil$.

Each batch is essentially a BBF, therefore the false positive probability for a batch can be found using Eqn. 1: $Pr_{fp\text{-}batch} = (1-(1-1/m_b)^{k_b x})^{mk_b/m_b}$, where x denotes the number of elements in the batch. For full batches, x equals to n_{thres} by construction, and for the last batch, x equals to $n-(\lambda-1)\times n_{thres}$. The cumulative false positive probability for the D-BBF structure is

$$\begin{aligned} Pr_{fp} =& 1-\prod_{i=1}^{\lambda}\left(1-Pr_{fp\text{-}batch}[\text{batch i}]\right) \\ =& 1-\left(1-(1-(1-1/m_b)^{k_b(n-(\lambda-1)n_{thres})})^{mk_b/m_b}\right)\times\prod_{i=1}^{\lambda-1}\left(1-(1-(1-1/m_b)^{k_b n_{thres}})^{mk_b/m_b}\right) \\ \leq& 1-\left(1-(1-(1-1/m_b)^{k_b n_{thres}})^{mk_b/m_b}\right)^{\lambda} \approx 1-\left(1-\left(1-e^{-k_b n_{thres}/m_b}\right)^{mk_b/m_b}\right)^{\lambda} \end{aligned} \quad (2)$$

Although Eqn. 2 is accurate, it is difficult to interpret. For a better insight on how the false positive probability grows with the number of blocks, we derive an upper bound as follows. A false positive event occurs when at least one of the batches in the D-BBF returns a false positive. Thus, the probability of a false positive is the probability of at least one batch to return a false positive, minus the probability of at least two batches to return a false positive, plus the probability of at least three batches to return a false positive, and so forth. The dominant term in this equation is the first term (the probability of at least one batch to return a false positive), and the true value of the equation is always less than the value of the first term. Therefore, the following inequality is valid: $Pr_{fp} \leq \sum_{i=1}^{\lambda} Pr_{fp\text{-}batch}[\text{batch i}] = \lambda\times\left(1-e^{-k_b n_{thres}/m_b}\right)^{mk_b/m_b}$. In the simplified inequality we clearly see that the false positive probability of the D-BBF structure grows at most linearly with λ.

Configuration of Dynamic Block-partitioned Bloom filters. The configuration has two objectives. First, it must yield a flexible D-BBF, which can be reduced effectively to address the particular requirements of the application, decided at runtime. Second, it must optimize the D-BBF structure so that it maintains a low false positive probability, even after the resolution reduction step.

The configuration step allows the application to impose the following constraints:

- The length of each batch of blocks m: This value will be chosen such that block processing and transmission can be performed efficiently. For example, each single batch can be set to 8 Kbytes, so that it can be always cached to the very fast L1 or L2 processor's cache (modern off-the-shelf PC's have at least 32 Kbytes L1 cache and around 1 Mbyte of L2 cache). Another option is to set it to the Maximum Transmission Unit (MTU) value, so that each batch of blocks can be packed to a single TCP/IP network message.
- The false positive probability per batch $Pr_{fp\text{-}batch}$: When the communication cost can be formalized as a trade-off between the false positive probability and the length of each batch, $Pr_{fp\text{-}batch}$ is set to the value that optimizes the trade-off. Otherwise, it can be the maximum false positive probability accepted by the user.

We now need to decide on the values of m_b, k_b, and n_{thres}. Similarly to the case of BBFs, we set k_b to 1 by default, because this value gives the maximum flexibility for the length reduction step without causing noticeable increase in the false positive probability. We then need to choose the values of m_b and n_{thres} that will minimize the overall false positive probability in the D-BBF. This probability can be minimized by maximizing the number of elements per batch n_{thres}, thereby reducing the number of batches required to represent the set. To find the value of m_b that maximizes n_{thres} we use local search. We first start with

a single block per batch of length $m_b = m$, and compute the maximum number of elements per batch n_{thres}, such that the false positive probability per batch is at most $Pr_{fp\text{-}batch}$. Recall that each batch is a standalone BBF, hence the maximum number of elements per batch can be computed efficiently using Eqn. 1. We then gradually increase the number of blocks per batch μ, adjusting the length per block to m/μ. For each μ, we compute the corresponding value of n_{thres}, keeping track of the μ which leads to the maximum n_{thres}. Note that we are only interested in the values of μ which satisfy the constraint $m/\mu \in \mathbb{N}_1$, thereby reducing the solution space to $O(\log(m))$ possible values. The cost of this computation is negligible, and it happens only during initialization. An additional optimization is possible by considering that the function that describes the relation of n_{thres} and μ is concave. Therefore, hill climbing optimization is guaranteed to derive the value of m_b that maximizes n_{thres}. An alternative method for deriving the optimal m_b is by assuming real values for n_{thres} and m_b, and using derivation to optimize the equation.

4.1 Evaluation

The purpose of the experiments was to examine the suitability of D-BBFs for representing sets of unknown cardinalities, such as streams and non-materialized sets. To systematically evaluate the structure, we initialized D-BBFs for an expected set cardinality, and used them to represent sets that were either larger or smaller than the expected set cardinality. We compared D-BBFs with standard Bloom filters, with respect to length and false positive probability.

We first chose the expected set cardinality $c \in \{50000, 100000, 200000\}$, and the target false positive probability $Pr_{fp} \in \{0.1, 0.05, 0.01, 0.005\}$. For each combination of c and Pr_{fp}, we initialized a standard Bloom filter, denoted as BF, and a Dynamic Block-partitioned Bloom filter, denoted as D-BBF. Following, we created a set S with cardinality $|S|$ in the range of $[c/5 \ldots 5c]$. Because the process of inserting elements in Bloom filters is based on hashing, the experimental results are orthogonal to the type of elements in S, which in our case was random integers. For example, the results also apply to stream summarization, where these sets would contain the elements of the stream, which could be of any type.

After initializing S, we added all elements of S to both BF and D-BBF. For the D-BBF structure, after adding all elements, we also reduced its resolution so that it offered a false positive probability closer to the chosen probability Pr_{fp}. The standard Bloom filter did not allow this resolution reduction since it lacks length flexibility. Finally, we measured the resulting false positive probability and the length of the two structures.

Figures 4 a.-d. plot the false positive probabilities in relation to the cardinality of the set, when the two structures are configured for $Pr_{fp} = 0.1, 0.05, 0.01, 0.005$ respectively, assuming an expected set cardinality of 100000. Table 2 provides further details for selected experimental configurations. The results for different expected set cardinalities ($c \in \{50000, 200000\}$) were analogous. The limitation of standard Bloom filters is clearly visible in these results: they are rendered useless when the actual cardinality of the set is notably higher than the expected set cardinality, e.g., three times as much. Interestingly, selecting a lower target false positive probability Pr_{fp} does not alleviate the problem of standard Bloom filters. For example, for 500000 elements, the false positive probability of the standard Bloom filter with $Pr_{fp} = 0.1$ is nearly equal to the corresponding false positive probability of the standard filter initialized for $Pr_{fp} = 0.005$ (approximately 0.8). An explanation for this behavior can be derived from the fact that for optimizing the false positive probability for an expected set cardinality, the number of hash functions is chosen such that

Set cardinality	Optimized for Pr_{fp} =0.05 D-BBF Length (Kbits)	D-BBF Prob	BF, 610Kbits Prob	Optimized for Pr_{fp} =0.005 D-BBF Length (Kbits)	D-BBF Prob	BF, 1078Kbits Prob
25000	305	0.022	≈**0**	404	0.005	≈**0**
50000	458	0.021	**0.006**	674	0.003	≈**0**
100000	1220	**0.050**	**0.050**	2155	**0.005**	**0.005**
150000	1220	**0.055**	0.145	2155	**0.005**	0.037
200000	1830	**0.097**	0.272	3232	**0.010**	0.117
300000	2440	**0.142**	0.531	4310	**0.015**	0.380
400000	3050	**0.185**	0.726	5388	**0.020**	0.635
500000	3660	**0.226**	0.847	6466	**0.025**	0.805

Table 2 False positive probabilities and length for Dynamic Block-partitioned Bloom filter and standard Bloom filters. The lowest false positive probability for each experiment is printed in bold.

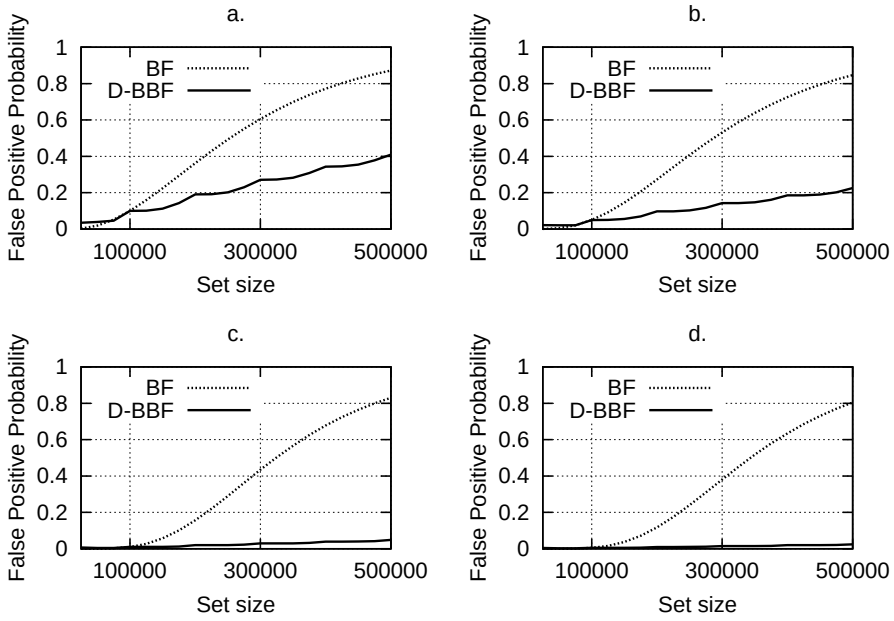

Fig. 4 False positive probabilities of D-BBF and standard Bloom filters. All structures are configured for 100000 elements and for initial false positive probability: a. 0.1, b. 0.05, c.0.01, d. 0.005.

the resulting filter has a density of 0.5. This is independent of the chosen false positive probability. Adding more than the anticipated elements in an optimized Bloom filter leads to a rapid increase of the density, causing more hash collisions and high false positive probabilities.

Regarding D-BBF, we first note that its false positive probability scales favorably with the cardinality of the set. Even the D-BBF configured for a relatively high initial false positive probability of 0.1 gives a maximum false positive probability of 0.4 for 500000 elements, whereas the corresponding false positive probability for the standard Bloom filter is over 0.8. The small fluctuation observed in the false positive probability of D-BBF (particularly visible in Figure 4 a.) indicates the addition of a new block in D-BBF. We also note that for large sets, false positive probability grows approximately linearly with the number of el-

ements, but with a very small coefficient, which is controlled from the initial false positive probability per batch. This property of D-BBF makes the structure suitable for representing sets of unknown cardinality.

As expected, D-BBFs are more accurate when they are initialized for a small false positive probability (compare for instance Figure 4 d. with Figure 4 a.). It is therefore beneficial to configure the D-BBF for a very low target false positive probability, and after all elements are added, to reduce its resolution to the minimum resolution that satisfies the required false positive probability.

It is also interesting to see how the length of D-BBFs compares with the length of standard Bloom filters (Table 2). Standard Bloom filters are configured for a fixed set cardinality, therefore their length remains constant throughout the experiment. We see that for small sets, large standard Bloom filters are inefficient; even though they reduce the false positive probability well below the targeted one, they require too much space, as they do not allow resolution reduction. For sets larger than the expected cardinality, standard Bloom filters require less space than D-BBFs, but they also have very high false positive probability. Instead, D-BBF structure works in a pay-as-you-go approach. It adds/removes batches and blocks, so that the false positive probability approximates the targeted false positive probability as much as possible.

Summarizing, D-BBFs have better scalability characteristics than standard Bloom filters, and they maintain an acceptable false positive probability even for sets significantly larger than the predicted ones. They also enable resolution reduction, which is important for distributed applications. These properties make the D-BBF structure a good choice for summarizing sets of unknown cardinalities for membership tests.

5 Cardinality Estimation for Bloom Filters

We now show how to estimate the cardinality of a set – the number of distinct elements it contains – based solely on its Bloom filter. This functionality is useful when it is too expensive to maintain or retrieve the full set, and only a Bloom filter representation of the set is available. This frequently occurs in stream processing [27], in Bloom joins [2, 3], and in other distributed systems. We describe several such applications and show how they can directly benefit from Bloom filter cardinality estimation in Section 6.

In Section 5.1 we describe cardinality estimation for standard Bloom filters. In Section 5.2 we show how the same principles are used to estimate the cardinality of set unions and intersections by using only the corresponding Bloom filters of the sets. We provide the corresponding analysis for Block-partitioned Bloom filters in Section 5.3. In Section 5.4 we present an extensive experimental evaluation for all proposed approaches.

5.1 Cardinality Estimation for standard Bloom Filters

We now show how to estimate the number of distinct elements hashed in a standard Bloom filter and derive probabilistic bounds for the estimation. Estimation requires only the number of true bits in the Bloom filter, and the Bloom filter configuration, i.e., number of hash functions and Bloom filter length. Briefly, the analysis proceeds as follows. With Lemma 1 we estimate the expected number of true bits in a Bloom filter, given that it contains n elements. This lemma is required for deriving the probabilistic bounds. Following, we estimate the number of elements in a Bloom filter, given the number of true bits. Finally, in Theorem 1

we derive probabilistic bounds for the number of elements added to a Bloom filter, given the number of true bits.

Lemma 1 *The expected number of true bits in a Bloom filter of length m with k hash functions after n elements were hashed is:* $\hat{S}(n) = m \times \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$. *Also, the following inequalities hold:*

Upper bound: *The probability of the number of true bits to be more than* $(1+\delta) \times \hat{S}(n)$ *is* $P(\#\ true\ bits > (1+\delta) \times \hat{S}(n)|n) \leq [e^{\delta}/(1+\delta)^{(1+\delta)}]^{\hat{S}(n)}$ *for* $\delta \geq 0$.

Lower bound: *The probability of the number of true bits to be less than* $(1-\delta) \times \hat{S}(n)$ *is* $P(\#\ true\ bits < (1-\delta) \times \hat{S}(n)|n) \leq e^{-\hat{S}(n)\delta^2/2}$ *for* $\delta \geq 0$.

Proof Given a Bloom filter of length m with k hash functions and n elements hashed into it. We define the binary random variables $Z_1, Z_2, \ldots Z_m$ where Z_i is interpreted to be the indicator variable for the event that the i^{th} bit in the Bloom filter is set to true. The probability that the i^{th} bit is set to true is $P[i = true] = 1 - \left(1 - \frac{1}{m}\right)^{kn}$. Having a Bloom filter of length m, the expected number of true bits equals to $\hat{S}(n) = \sum_{i=1}^{m} P[i = true] = m \times \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$. The bounds follow directly from the Chernoff inequality, by assuming (as is standard in the analysis of Bloom filters) that the random variables $Z_1, Z_2, \ldots Z_m$ are independent. For the lower bound, we use the simplified form proposed in [28], pp. 69–70. □

We now estimate the number of elements hashed in a Bloom filter based on the number of true bits in the Bloom filter t. We denote by $\hat{S}^{-1}(t)$ the inverse of $\hat{S}(n)$, so that $\hat{S}^{-1}(t)$ returns the number of elements that would result on an expected number of t true bits in the Bloom filter. We find $\hat{S}^{-1}(t)$ using the probability of a bit to be true:

$$P[i = true] = \frac{t}{m} = 1 - \left(1 - \frac{1}{m}\right)^{k\hat{S}^{-1}(t)} \Rightarrow \left(1 - \frac{1}{m}\right)^{k\hat{S}^{-1}(t)} = 1 - \frac{t}{m} \Rightarrow$$

$$\hat{S}^{-1}(t) = \frac{\ln\left(1 - \frac{t}{m}\right)}{k \times \ln\left(1 - \frac{1}{m}\right)} \tag{3}$$

$\hat{S}^{-1}(t)$ is the maximum likelihood value for the number of hashed elements given the state of the Bloom filter, and can be used as a rough estimate when a single number of hashed elements is required. However, strict error margins can only be derived for intervals of set cardinalities. The following theorem provides for a given interval the probability that the real cardinality of the set is indeed within the given bounds.

Theorem 1 *Given a Bloom filter BF of length m with k hash functions and t bits set to true. Let* $\hat{S}^{-1}(t)$ *denote the expected number of elements in a Bloom filter, given that the Bloom filter has t bits set to true. Then, for any* n_l, n_r *such that* $n_l \leq \hat{S}^{-1}(t-1)$ *and* $\hat{S}^{-1}(t+1) \leq n_r$, *the number of elements hashed in BF lies in the range* (n_l, n_r) *with a probability of at least* $1 - e^{t-1-\hat{S}(n_l)} \times [\hat{S}(n_l)/(t-1)]^{(t-1)} - e^{-\frac{(t+1-\hat{S}(n_r))^2}{2\hat{S}(n_r)}}$.

Proof If the number of elements is $n \leq n_l$, then $P(\#\ true\ bits \geq t|n) \leq P(\#\ true\ bits \geq t|n_l)$. Choosing n_l and δ_l such that $(1+\delta_l) \times \hat{S}(n_l) < t$, we obtain by Lemma 1 that this probability is $P(\#\ true\ bits > (1+\delta_l) \times \hat{S}(n_l)|n_l) \leq [e^{\delta_l}/(1+\delta_l)^{(1+\delta_l)}]^{\hat{S}(n_l)}$. Similarly, if the number of elements is $n \geq n_r$, then $P(\#\ true\ bits \leq t|n) \leq P(\#\ true\ bits \leq t|n_r)$. Choosing n_r and δ_r

such that $(1-\delta_r)\times\hat{S}(n_r) > t$, we obtain by lemma 1 that this probability is $P(\#\ true\ bits < (1-\delta_r)\times\hat{S}(n_r)|n_r) \leq e^{-\hat{S}(n_r)\delta_r^2/2}$.

For choices of $n_l, \delta_l, n_r, \delta_r$ as described above, we get that with probability $1-[e^{\delta_l}/(1+\delta_l)^{(1+\delta_l)}]^{\hat{S}(n_l)} - e^{-\hat{S}(n_r)\delta_r^2/2}$, the number of elements is in the range (n_l, n_r). We compute a value for δ_l such that $(1+\delta_l)\times\hat{S}(n_l) < t$. Clearly, $\delta_l = \frac{t-1-\hat{S}(n_l)}{\hat{S}(n_l)}$ satisfies the inequality. Similarly, we compute a value for δ_r such that $(1-\delta_r)\times\hat{S}(n_r) > t$. Clearly, $\delta_r = \frac{\hat{S}(n_r)-t-1}{\hat{S}(n_r)}$ satisfies the inequality.

We conclude that with probability of at least $1-e^{-\frac{(t+1-\hat{S}(n_r))^2}{2\hat{S}(n_r)}} - e^{(t-1-\hat{S}(n_l))}\times[\hat{S}(n_l)/(t-1)]^{(t-1)}$, the number of elements is in the range (n_l, n_r). □

Theorem 1 enables computing the cardinality of a set based on its Bloom filter representation, and in particular, on the Bloom filter length, number of hash functions, and number of bits set to true. It also allows computing of upper and lower cardinality bounds, for settings where probabilistic guarantees are important. As our experimental evaluation shows (Section 5.4), for Bloom filters of reasonable densities, i.e., that can be used for membership tests, estimations computed with Theorem 1 are highly accurate and probabilistic bounds are tight.

5.2 Cardinality Estimation for Bloom Filter Union and Intersection

Distributed algorithms frequently need to estimate the cardinality of a union or intersection of remote sets, having only the Bloom filters corresponding to the sets. For example, a query planner for a distributed database may need to estimate the cardinality of an equi-join or of a union of two remote tables, to devise the optimal query execution plan. Similarly, peers in a P2P network may need to coordinate for executing a query, by exchanging their Bloom filters. We describe these and other scenarios in detail in Section 6.

With respect to cardinality of the union, we note that a filter produced by bitwise-OR merging of the Bloom filters $BF_1, BF_2, \ldots, BF_n$ (the Bloom filters of sets $S_1, S_2, \ldots, S_n$) is identical to the Bloom filter of the set $S_\cup := S_1 \cup S_2 \cup \ldots \cup S_n$. Therefore, we estimate the cardinality of the set $S_\cup$ by applying Theorem 1 on the bitwise-OR produced Bloom filter.

Estimating the cardinality of the intersection of two or more sets from their Bloom filters is slightly more complicated, because merging with bitwise-AND does not result to a standard Bloom filter on which Theorem 1 can be applied. In particular, the same bits may have been set in the two individual Bloom filters BF_1 and BF_2 from two different elements, the one belonging only to the first set and the other belonging only to the second set. These bits will be incorrectly set to true in the AND-merged Bloom filter, i.e., the Bloom filter produced by merging BF_1 and BF_2 with bitwise-AND. Therefore, the resulting density of the AND-merged Bloom filter will not be representative of the cardinality of the intersection of the two sets.

The probability for such a bit collision can be high, especially in dense Bloom filters. Consider for instance two sets S_1 and S_2, created by randomly selecting elements from a very large universe of elements. Let BF_1 and BF_2 be the Bloom filters of the two sets, both with length m and k hash functions. $BF_\wedge$ represents the Bloom filter produced by bitwise-AND merging of BF_1 and BF_2. Then, the probability for each bit to be set in BF_1 and BF_2 from two different elements, thus also falsely be in $BF_\wedge$ is: $(1-(1-\frac{1}{m})^{kn_1})\times(1-(1-\frac{1}{m})^{kn_2})$. For dense Bloom filters, this probability is high and can significantly influence the density of $BF_\wedge$, and thus falsify our previous cardinality estimation function. However, we can use the

density of the initial Bloom filters for estimating the number of these random bit collisions (we refer to the number of these collisions as r_{bits}). Then, we can subtract r_{bits} from the number of true bits in $BF_\wedge$, and use this number for the estimation of the cardinality of $S_1 \cap S_2$.

For the analysis we use the following notations: $S_{1\cap 2}$ denotes the intersection of sets S_1 and S_2. The Bloom filters of the sets are denoted with BF_1, BF_2 and $BF_\cap$. With $BF_\wedge$ we refer to the Bloom filter produced by merging BF_1 and BF_2 with bitwise-AND. Finally, t_x refers to the number of true bits in Bloom filter BF_x, e.g., $t_\wedge$ denotes the count of true bits in $BF_\wedge$.

Lemma 2 *Let BF_1, BF_2, and $BF_\cap$, denote the Bloom filters of S_1, S_2 and $S_1 \cap S_2$ respectively. All filters have length m and use the same k hash functions. $BF_\wedge$ denotes the Bloom filter created by bitwise AND of BF_1 and BF_2. The expected number of bits that are set in $BF_\wedge$ but are not set in $BF_\cap$ is $\hat{r}_{bits} = \frac{(t_1-t_\cap)\times(t_2-t_\cap)}{m-t_\cap}$.*

Proof For the proof we represent Bloom filters as a set of numbers, so that $if f(BF[i] = true)$ then $i \in SET_{BF[i]}$. By definition of r_{bits}:

$$|SET_{BF_\wedge}| = |SET_{BF_\cap}| + r_{bits}$$

where $|SET_x|$ denotes the cardinality of SET_x. Assuming that the hash functions in each Bloom filter are independent (a standard assumption for Bloom filters), the elements in $SET_{BF_1} \backslash SET_{BF_\cap}$ are independent from the elements in $SET_{BF_2} \backslash SET_{BF_\cap}$. Thus the probability of a number to occur in both $SET_{BF_1} \backslash SET_{BF_\cap}$ and $SET_{BF_2} \backslash SET_{BF_\cap}$ is $\frac{|SET_{BF_1}|-|SET_{BF_\cap}|}{m-|SET_{BF_\cap}|} \times \frac{|SET_{BF_2}|-|SET_{BF_\cap}|}{m-|SET_{BF_\cap}|}$.

This gives a maximum likelihood value for r_{bits}:

$$\begin{aligned}\hat{r}_{bits} &= (m - |SET_{BF_\cap}|) \times \frac{|SET_{BF_1}|-|SET_{BF_\cap}|}{m-|SET_{BF_\cap}|} \times \frac{|SET_{BF_2}|-|SET_{BF_\cap}|}{m-|SET_{BF_\cap}|} \\ &= \frac{(t_1-t_\cap)\times(t_2-t_\cap)}{m-t_\cap}\end{aligned} \tag{4}$$

□

Similar to the analysis for standard Bloom filters, we define a function for estimating the number of true bits in the Bloom filter $BF_\wedge$, assuming that the number of elements in the intersection $|S_1 \cap S_2|$ is known.

Lemma 3 *Let BF_1 and BF_2 be the Bloom filters of sets S_1 and S_2 respectively. The Bloom filters have length m and share the same k hash functions. $BF_\wedge$ is the Bloom filter created by a bitwise AND of BF_1 and BF_2. Then, the function $\hat{S}(t_1, t_2, n_\cap) := \frac{t_1\times t_2 + m\times(1-(1-1/m)^{kn_\cap})\times(m-t_1-t_2)}{m\times(1-1/m)^{k\times n_\cap}}$ returns the expected number of bits that are set in $BF_\wedge$, where $n_\cap$ denotes the cardinality of $S_1 \cap S_2$, and t_x denotes the count of the true bits set in the Bloom filter BF_x. Also the following inequalities hold:*

Upper bound: *The probability that the number of true bits in $BF_\wedge$ is more than $(1+\delta)\times\hat{S}(t_1, t_2, n_\cap)$ is $P(\#\ true\ bits > (1+\delta)\times\hat{S}(t_1, t_2, n_\cap)|n) \leq [e^\delta/(1+\delta)^{(1+\delta)}]^{\hat{S}(t_1,t_2,n_\cap)}$ for $\delta \geq 0$.*

Lower bound: *The probability that the number of true bits in $BF_\wedge$ is less than $(1-\delta)\times\hat{S}(t_1, t_2, n_\cap)$ is $P(\#\ true\ bits < (1-\delta)\times\hat{S}(t_1, t_2, n_\cap)) \leq e^{-\hat{S}(t_1,t_2,n_\cap)\times\delta^2/2}$ for $\delta \geq 0$.*

Proof

$$\begin{aligned} t_\wedge =& t_\cap + r_{bits} \\ =& t_\cap + \frac{(t_1 - t_\cap) \times (t_2 - t_\cap)}{m - t_\cap} \\ =& \frac{t_1 \times t_2 + t_\cap \times (m - t_1 - t_2)}{m - t_\cap} \\ =& \frac{t_1 \times t_2 + m \times \left(1 - (1 - 1/m)^{kn_\cap}\right) \times (m - t_1 - t_2)}{m - m \times \left(1 - (1 - 1/m)^{kn_\cap}\right)} \\ =& \frac{t_1 \times t_2 + m \times \left(1 - (1 - 1/m)^{kn_\cap}\right) \times (m - t_1 - t_2)}{m \times (1 - 1/m)^{kn_\cap}} \end{aligned} \tag{5}$$

The bounds follow directly from Chernoff inequalities, as in Lemma 2. □

Next, we estimate the number of elements in the intersection $S_1 \cap S_2$ from the Bloom filters BF_1, BF_2 and $BF_\wedge$. We denote by $\hat{S}^{-1}(t_1, t_2, t_\wedge)$ the inverse of $\hat{S}(t_1, t_2, n_\cap)$, so that given t_1, t_2 and $t_\wedge$, function $\hat{S}^{-1}(t_1, t_2, t_\wedge)$ returns the expected cardinality of $S_1 \cap S_2$. Similar to the analysis for the single Bloom filter, we can find $\hat{S}^{-1}(t_1, t_2, t_\wedge)$ using the probability of a bit to be true in $BF_\wedge$:

$$\begin{aligned} P(i = true) =& \frac{t_\wedge}{m} \\ =& \frac{t_1 \times t_2 + m \times \left(1 - (1 - 1/m)^{k \times \hat{S}^{-1}(t_1, t_2, t_\wedge)}\right) \times (m - t_1 - t_2)}{m^2 - m^2 \times \left(1 - (1 - 1/m)^{k \times \hat{S}^{-1}(t_1, t_2, t_\wedge)}\right)} \Rightarrow \\ \hat{S}^{-1}(t_1, t_2, t_\wedge) =& \frac{\ln\left(m - \frac{t_\wedge \times m - t_1 \times t_2}{m - t_1 - t_2 + t_\wedge}\right) - \ln(m)}{k \times \ln(1 - 1/m)} \end{aligned} \tag{6}$$

$\hat{S}^{-1}(t_1, t_2, t_\wedge)$ is the most likely number of elements in $S_1 \cap S_2$. Similar to the normal Bloom filter cardinality estimation (Theorem 1), we can set upper and lower bounds for the estimation of $\hat{S}^{-1}(t_1, t_2, t_\wedge)$. The following theorem provides for a given interval the probability that the real cardinality of $S_1 \cap S_2$ is indeed within the given bounds.

Theorem 2 *Let BF_1 and BF_2 be the Bloom filters of S_1 resp. S_2, with length m and k hash functions. $BF_\wedge$ refers to the Bloom filter produced by bitwise AND of BF_1 and BF_2, and with t_x we denote the count of the true bits set in the Bloom filter BF_x. For any n_l, n_r such that $n_l < \hat{S}^{-1}(t_1, t_2, t_\wedge) < n_r$, the number of elements in the intersection $S_1 \cap S_2$ lies in the range (n_l, n_r) with probability of at least*

$$1 - \left(\frac{\hat{S}(t_1, t_2, n_l)}{t_\wedge - 1}\right)^{t_\wedge - 1} e^{(t_\wedge - 1 - \hat{S}(t_1, t_2, n_l))} - e^{-\frac{(t_\wedge + 1 - \hat{S}(t_1, t_2, n_r))^2}{2 \times \hat{S}(t_1, t_2, n_r)}}$$

Proof Via Chernoff bounds, similar to the proof for Theorem 1.

Theorem 2 does not directly hold for Bloom filters created by the intersection of more than two Bloom filters. It is not possible to derive closed-form equations for the Theorem which address an arbitrary number of Bloom filters, and therefore we do not present this analysis here. Nevertheless, the corresponding equations can be extended for individual cases, following the example for the 2 Bloom filters.

5.3 Cardinality Estimation for Block-partitioned Bloom filters

The proposed cardinality estimation approach presented in Section 5.1 can also be used for Block-partitioned Bloom filters, i.e., by considering one of the blocks to estimate the cardinality of the BBF. However, we can get more accurate cardinality estimation and stricter bounds if we account for all λ blocks in the Block-partitioned filter. Since the analysis is very similar to the analysis presented at Section 5.1, we only present sketches of the proofs.

First we find the expected number of true bits in BBF after n elements are hashed in the Bloom filter.

Lemma 4 *The expected number of true bits in a Bloom filter with λ blocks, each of length m_b and with k_b hash functions after n elements were hashed is: $\hat{S}(n) = \lambda m_b \left(1 - (1 - \frac{1}{m_b})^{k_b n}\right)$. Also, the following inequalities hold:*

Upper bound: *The probability of the number of true bits to be more than $(1+\delta)\times \hat{S}(n)$ is $P(\#\ true\ bits > (1+\delta)\times\hat{S}(n)|n) \leq [e^{\delta}/(1+\delta)^{(1+\delta)}]^{\hat{S}(n)}$ for $\delta \geq 0$.*

Lower bound: *The probability of the number of true bits to be less than $(1-\delta)\times \hat{S}(n)$ is $P(\#\ true\ bits < (1-\delta)\times\hat{S}(n)|n) \leq e^{-\hat{S}(n)\delta^2/2}$ for $\delta \geq 0$.*

Sketch Each block in a BBF is an independent Bloom filter. Therefore, the expected number of true bits in a block which contains n elements can be found with Lemma 2, and it is $\hat{S}_{block}(n) = m_b(1 - (1 - \frac{1}{m_b})^{k_b n})$. Since the Block-partitioned Bloom filter has λ blocks, the expected number of true bits in all blocks is $\hat{S}(n) = \lambda \times \hat{S}_{block}(n) = \lambda m_b \left(1 - (1 - \frac{1}{m_b})^{k_b n}\right)$. Bounds follow directly from Chernoff bounds. □

The following theorem provides for a given interval the probability that the cardinality of the BBF is within this interval.

Theorem 3 *Given a Block-partitioned Bloom filter BBF with λ blocks, and t bits set to true. Each of the blocks has length m_b and k_b hash functions. Then, the expected number of distinct elements hashed in BBF is $\hat{S}^{-1}(t) = \frac{\log(1-t/(\lambda m_b))}{k_b \log(1-1/m_b)}$. Furthermore, for any n_l, n_r such that $n_l \leq \hat{S}^{-1}(t-1)$ and $\hat{S}^{-1}(t+1) \leq n_r$, the number of elements hashed in the Bloom filter lies in the range (n_l, n_r) with a probability of at least $1 - e^{(t-1-\hat{S}(n_l))} \times [\hat{S}(n_l)/(t-1)]^{(t-1)} - e^{-\frac{(t+1-\hat{S}(n_r))^2}{2\hat{S}(n_r)}}$.*

Sketch Let $\hat{S}^{-1}(t)$ denote the number of elements that would result on an expected number of t true bits in BBF. The probability of a random bit i from BBF to be set to true is

$$P[i = true] = t/(\lambda m_b) = 1 - (1 - 1/m_b)^{k_b \hat{S}^{-1}(t)} \Rightarrow$$
$$\hat{S}^{-1}(t) = \frac{\log(1 - \frac{t}{\lambda m_b})}{k_b \log(1 - 1/m_b)}$$

Bounds follow directly from Chernoff bounds, as in Theorem 1. □

5.4 Evaluation

While our analysis already offers probabilistic bounds for cardinality estimation, we also evaluated experimentally the influence of Bloom filter length, number of hash functions, and number of blocks, on estimation accuracy. The experiments covered all three cases, standard Bloom filters, Bloom filter intersection, and Block-partitioned Bloom filters.

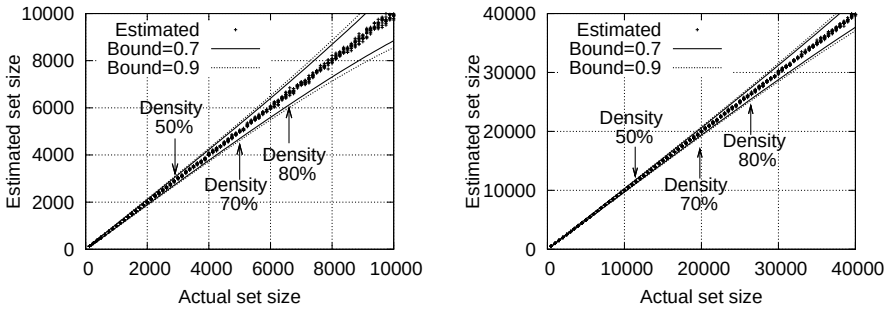

Fig. 5 Estimation accuracy for standard Bloom filters: a. 8192 bits, 2 hash functions, b. 32 Kbits, 2 hash functions.

All experiments shared the same experimental setup, which we present here. Then, in the following sections we present and discuss the results for the three structures separately. Based on the experimental results, we also point out some practical considerations with respect to the optimal choice of Bloom filter configuration for cardinality estimation.

For each individual experiment, we first generated a set S of cardinality c, and created its (Block-partitioned) Bloom filter representation. Using the filter representation, we estimated the cardinality of S according to the theorems presented earlier in this section, and evaluated the accuracy of the estimation. For the case of AND-merged Bloom filters, we generated an additional set S', having a pre-configured overlap with S, i.e., $|S \cap S'| = ovl$. We then estimated the cardinality of $|S \cap S'|$ using the AND-merged Bloom filters of the two sets, and evaluated the accuracy of the estimation.

To assess the influence of different factors, we varied the Bloom filter length, the cardinality c, and the number of hash functions. For the case of Block-partitioned Bloom filters, we also varied the number of Blocks, whereas for the AND-merged Bloom filters we varied the size of the overlap. For each setting, we repeated the experiment multiple times to even out random effects. We report average, maximum, and standard deviation of the actual relative error after 1000 repetitions, and analytical bounds for the probabilities 0.7 and 0.9.

In the next section we describe and discuss our findings for standard Bloom filters. Results corresponding to AND-merged Bloom filters are presented in Section 5.4.2. Section 5.4.3 discusses the evaluation results for Block-partitioned Bloom filters.

5.4.1 Cardinality Estimation for Bloom Filters

We evaluated cardinality estimation for standard Bloom filters varying the Bloom filter length between 8192 bits and 8192 Kbits, and the number of hash functions between 1 and 10. For each experimental configuration, we have set the maximum cardinality of S to the one that resulted to a Bloom filter of density 0.9. Cardinality estimation and the respective probabilistic bounds were computed using Theorem 1.

Influence of set cardinality. Figure 5 plots the relation between the actual and estimated cardinality. Each experiment repetition is marked as a dot; for clarity, only the first 10 repetitions are included in the figure. We also include the probabilistic bounds for each setting for comparison purposes. The results are for Bloom filters of length 8 Kbits and 32 Kbits, each with 2 hash functions. To show how Bloom filter density affects estimation accuracy, we have also marked the points where the density of the Bloom filter reaches 50%, 70%, and 80%. Table 3 presents further details for some selected configurations, after 1000 repetitions.

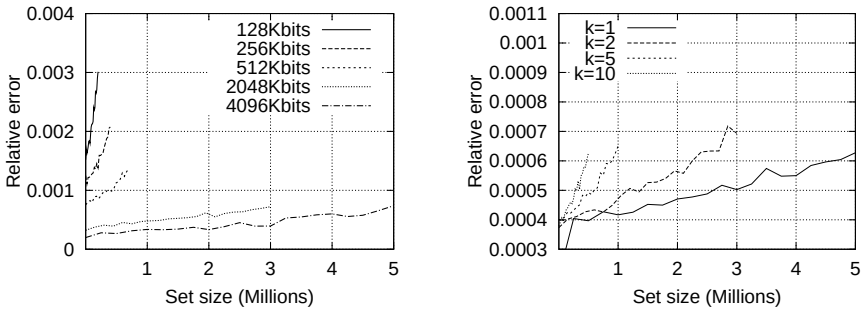

Fig. 6 Estimation accuracy for standard Bloom filters: a. Influence of Bloom filter length, b. Influence of number of hash functions.

We see that estimations are always very close to the actual Bloom filter cardinalities. This is observed for all Bloom filter lengths. In fact, maximum observed relative error was only 0.04, and it occurred for the Bloom filter of 8 Kbits, when density was already 0.9 (cf. Table 3). Note that the corresponding Bloom filter with this density was already useless for membership tests, as it exhibited a false positive probability of 0.81. We also see that the probabilistic bounds are very tight for moderate densities, i.e., densities up to 0.7. Standard deviation of the relative error is also small, and the maximum error is close to the average error. The same outcome is observed over the whole range of tested Bloom filter lengths and densities.

We also see that our analysis offers an appropriate and efficient approach for counting the distinct elements in huge sets. For example, a Bloom filter of 4096 Kbits (0.5 Mbyte) with 2 hash functions is already sufficient for accurately estimating the cardinalities of sets having as much as 5 million distinct elements (Figure 6 a.). For efficiently counting 10 million distinct elements, a Bloom filter of 1 Mbyte with 2 hash functions is sufficient (Table 3).

Influence of Bloom filter length. Figure 6 a. plots the average relative error with respect to the number of elements for Bloom filters of various lengths, all with 2 hash functions. We see that for the same number of elements, a larger Bloom filter yields higher estimation accuracy, as expected. Furthermore, probabilistic bounds are tighter, and the standard deviation of relative error is lower for larger Bloom filters.

Table 3 also shows that larger Bloom filters exhibit higher accuracy for the same density. For example, for a density of 0.534, the average relative error for a filter of 256 Kbits is 0.0013, compared to 0.0009 for the filter of 512 Kbits with the same density. The same effect is observed with respect to maximum relative error and standard deviation. Moreover, probabilistic bounds are tighter in the larger Bloom filters. The practical significance of this result is that applications for which tight probabilistic bounds are important, should create larger Bloom filters so that they maintain low density. Counting the elements of a stream is one such application for which tight probabilistic bounds may be desired. On the other hand, when a rough cardinality estimation is sufficient, e.g., when cardinality is used to optimize Bloom joins, a smaller Bloom filter can also be used.

Influence of number of hash functions. Figure 6 b. shows the relative errors for Bloom filters of length 2048 Kbits, for different numbers of hash functions. As expected, the number of hash functions also affects estimation accuracy. This effect is indirect, via density: by increasing the number of hash functions, the resulting density increases, causing a higher relative error. Therefore, Bloom filters with less hash functions can accommodate more el-

Length (Kbits)	Hashes	Items	Density	Relative error			Probabilistic bounds
				Avg.	Max.	Std.dev.	
Only number of elements varies							
8	2	3000	0.519	7.2E-3	0.032	5.77E-3	**70%:**2837–3161 **90%:**2793–3204
8	2	9000	0.889	9.44E-3	0.040	7.23E-3	**70%:**8103–9887 **90%:**7879–10111
Bloom filter length varies							
256	2	100000	0.534	1.34E-3	5.37E-3	1.02E-3	**70%:**99044–100955 **90%:**98764–101235
256	2	200000	0.783	1.44E-3	6.38E-3	1.06E-3	**70%:**197523–202475 **90%:**196808–203190
512	2	200000	0.534	8.76E-4	3.35E-3	6.66E-4	**70%:**198647–201350 **90%:**198250–201747
8192	2	1E7	0.908	1.13E-3	2.2E-3	4.04E-4	**70%:**9964361–10035632 **90%:**9953888–10046105
Number of hash functions varies							
2048	1	500000	0.212	3.65E-4	1.64E-3	2.8E-4	**70%:**497123–502875 **90%:**496386–503611
2048	10	500000	0.908	6.46E-4	2.96E-3	4.91E-4	**70%:**496439–503559 **90%:**495404–504594

Table 3 Estimation accuracy for standard Bloom filters.

ements, and can offer a more accurate cardinality estimation. Therefore, with respect to cardinality estimation, using a single hash function is the best choice.

As explained in Section 2, the optimal number of hash functions for membership test is the one which achieves a density closest to 0.5. This seems to imply a trade-off between optimizing a Bloom filter for membership tests and for cardinality estimation. However, as long as the density of the Bloom filter stays on a level *acceptable for membership tests*, the effect of the number of hash functions in cardinality estimation accuracy is negligible. Thus, applications that use the Bloom filters for both membership tests and cardinality estimation should select the number of hash functions such that false positive probability for membership test is minimized. Applications that use Bloom filters solely for cardinality estimation should use only one hash function, which is the optimal number with respect to cardinality estimation.

In summary, experimental results confirm the suitability of Theorem 1 for estimating Bloom filter cardinality. They show that cardinality estimation is highly accurate, even for very dense Bloom filters and that probabilistic bounds are tight for moderate Bloom filter densities.

5.4.2 Cardinality Estimation for Bloom Filter Intersection

For evaluating experimentally the estimation accuracy of Theorem 2 we have constructed sets with various cardinalities and intersection ratios, and used their Bloom filters to estimate the cardinality of their intersection. Similar to the previous experiments, the evaluation was repeated for multiple Bloom filter configurations, with Bloom filter lengths in the range of 8192 bits to 8192 Kbits, and with 1 to 10 hash functions.

Unless otherwise noted, the following results correspond to the configuration where the the two sets S and S' have the same cardinaltiy c, and an intersection of $c/2$ elements. For each Bloom filter configuration, the maximum cardinality of the sets was set to the one that results to a filter with density 0.9.

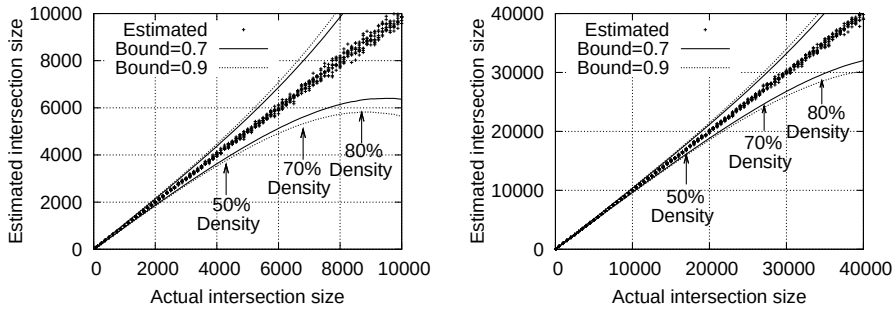

Fig. 7 Estimation accuracy for Bloom filter Intersection: a. 16 Kbits, 2 hash functions, b. 64 Kbits, 2 hash functions.

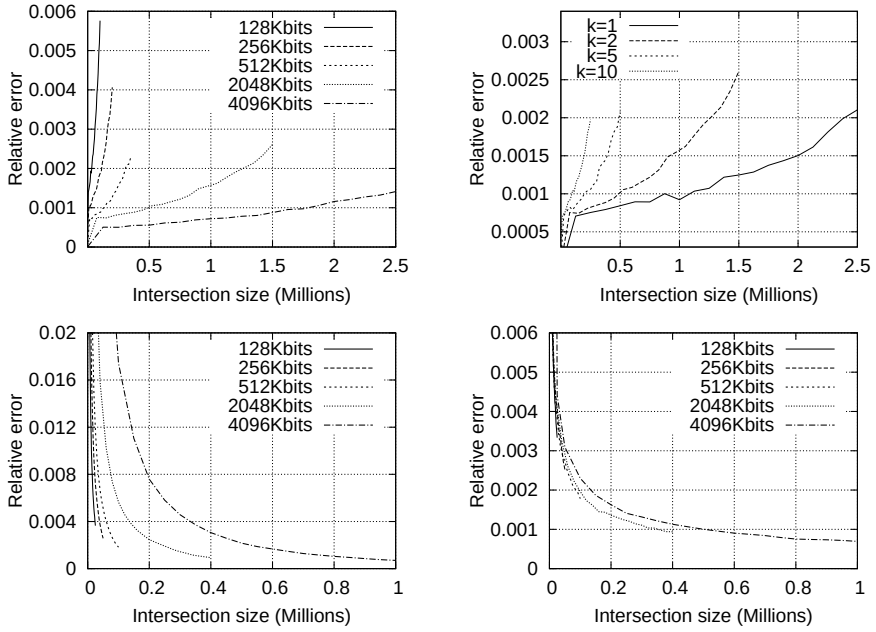

Fig. 8 Estimation accuracy for Bloom filter Intersection: a. Varying the cardinalities of both sets, b. Varying the number of hash functions, c. Varying the overlap ratio, d. Varying the cardinality of only one set.

Influence of the set cardinalities. Figure 7 plots the estimated intersection cardinality in correlation to the actual intersection cardinality, for Bloom filters of 16 and 64 Kbits with 2 hash functions. Similar to the figures for standard Bloom filters, this figure is also annotated with probabilistic bounds and density marks. In Table 4 we present detailed results for sample configurations.

The experimental results for Bloom filter intersection are similar to the results for standard Bloom filters. Cardinality estimation is highly accurate for Bloom filters of reasonable densities, i.e., which could also be used for the purpose of membership tests. Also, the observed maximum relative error and the standard deviation are very small, and probabilistic bounds are tight. Therefore, an application which uses AND-merged Bloom filters for mem-

Length (Kbits)	Hashes	$\lvert S \cap S' \rvert$	Density	Relative error			Probabilistic bounds
				Avg.	Max.	Std.dev.	
Only number of elements varies							
8	2	2000	0.478	0.016	0.060	0.012	**70%:**1736–2262 **90%:**1670–2329
8	2	5000	0.852	0.034	0.177	0.026	**70%:**2606–7379 **90%:**2162–7823
Bloom filter length varies							
256	2	50000	0.386	2.6E-3	0.012	1.95E-3	**70%:**49022–50974 **90%:**48774–51223
256	2	100000	0.667	3.64E-3	0.014	2.67E-3	**70%:**96024–103967 **90%:**95020–104970
512	2	100000	0.386	1.86E-3	8.39E-3	1.41E-4	**70%:**98620–101376 **90%:**98267–101730
8192	2	5E6	0.844	1.02E-3	4.05E-3	7.6E-4	**70%:**4908341–5091644 **90%:**4885133–5114852
Number of hash functions varies							
2048	1	250000	0.124	7.07E-4	2.94E-3	5.39E-4	**70%:**248576–251423 **90%:**248211–251788
2048	10	250000	0.844	1.91E-3	8.57E-3	1.5E-3	**70%:**240923–259072 **90%:**238656–261338

Table 4 Estimation accuracy for Bloom filter intersection.

bership testing already achieves high cardinality estimation accuracy. Nevertheless, average relative errors are still low even even for extremely dense AND-merged Bloom filters which would otherwise be considered useless for membership testing.

Influence of Bloom filter length. In Figure 8 a. we plot the average relative error for Bloom filters of various lengths, all with 2 hash functions. For the same set of elements, the estimation clearly becomes more accurate by increasing the Bloom filter length. Similarly, for a fixed density, relative error reduces when Bloom filter length increases (as an example compare the results for 256 Kbits and 512 Kbits filters in Table 4). Bloom filter length also affects standard deviation and tightness of probabilistic bounds: by increasing the length, probabilistic bounds get tighter and standard deviation of relative error is reduced. Therefore, an application developer can choose arbitrarily tight probabilistic bounds by increasing the length of the Bloom filters.

Influence of number of hash functions. Figure 8 b. presents the average relative error for AND-merged Bloom filters of 2048 Kbits, with different numbers of hash functions. We see that estimation accuracy improves when the number of hash functions is reduced, for the same reason as with the standard Bloom filter. Nevertheless, the difference between relative errors is negligible when density of the AND-merged filter is in levels acceptable for membership testing. Thereby, Bloom filters which are optimized for membership tests already offer significantly accurate cardinality estimation, very close to the optimal one.

Influence of intersection characteristics. In the previous experiments, the two sets S and S' were always constructed with the same cardinality c and with an overlap of $c/2$. To evaluate the generic applicability of Theorem 2, we also conducted experiments with sets of different overlap ratios, and of different cardinalities.

Figures 8 c. and d. plot the average relative error corresponding to the cardinality of the intersection, for the two different cases. For Figure 8 c. the two sets were generated with the same cardinality c, and with an intersection cardinality in the range of $[0 \dots c]$. The cardinality of the two sets was set to the one leading to Bloom filters of density 0.5. For Figure 8 d., S was generated with a fixed cardinality c, whereas S' was generated with

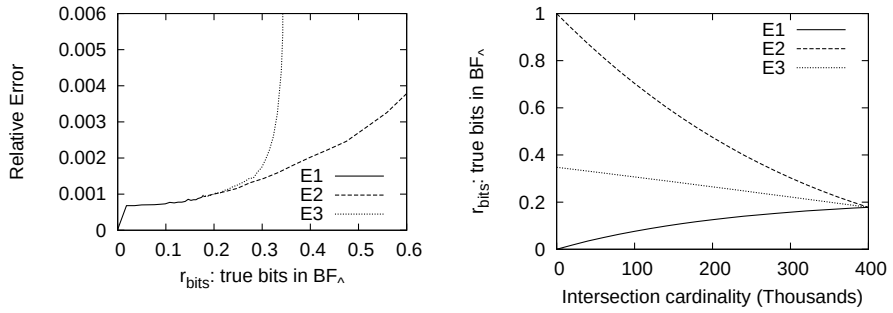

Fig. 9 a. Influence of the ratio of r_{bits} to true bits to the relative error, b. Ratio of r_{bits} to true bits for the different configurations

a cardinality c' in the range of $[0 \dots c]$, and with an intersection of $c'/2$ elements with S. Table 5 presents detailed results for sample configurations, focusing on the Bloom filter of 2048 Kbits, with 2 hash functions.

We observe that for both experimental setups, relative error is reduced with an increase in the intersection cardinality. This observation is consistent for all Bloom filter lengths. This is an interesting result, since at first sight it appears to be contradicting to the results reported earlier (e.g., Figures 8 a. and b.), where the relative error is increased with the cardinality of the intersection. An indication for why this happens is derived from Figure 9 a., which shows the relative error corresponding to $r_{bits} : tb(BF_\wedge)$, i.e., the ratio of r_{bits} to true bits in the AND-merged Bloom filter. The figure includes the results for the three experimental setups described earlier, denoted as follows:

- **[E1:]** Varying the cardinalities of both sets, with $|S| = |S'| = c$, and $|S \cap S'| = c/2$ (Figure 8 a.).
- **[E2:]** Keeping the cardinalities of both sets fixed and equal, and varying the overlap $|S \cap S'|$ in the range $[0 \dots c/2]$ (Figure 8 c.).
- **[E3:]** Keeping the cardinality of S fixed and varying the cardinality of S'. The overlap is set to $|S'|/2$ (Figure 8 d.).

We see that the ratio $r_{bits} : tb(BF_\wedge)$ determines the relative error. For E1, increasing the intersection cardinality leads to an increase of this ratio, because the r_{bits} increase more rapidly than the true bits in the Bloom filter intersection (cf. Fig. 9 b.). On the other hand, for experiments E2 and E3, increasing the intersection cardinality results to a decrease in this ratio. In particular, for E2, the r_{bits} constantly decrease with an increase of the overlap, thereby decreasing the ratio of $r_{bits} : tb(BF_\wedge)$. For E3, r_{bits} and true bits increase in parallel, but the true bits increase in a faster rate, which results in a decreasing ratio. The relative error decreases with this ratio.

Summarizing the experimental results, Theorem 2 estimates with high accuracy the cardinality of the intersection of the two sets using their Bloom filter representations. Tightness of the probabilistic bounds depends on the density of the two Bloom filters, but even for high densities, the probabilistic bounds are sufficiently tight for practical concerns.

5.4.3 Cardinality estimation for Block-partitioned Bloom Filters

For the evaluation of cardinality estimation for Block-partitioned Bloom filters, we varied the number of blocks between 1 and 10, and the block length between 4 and 4096 Kbits.

$\|S\|$	$\|S'\|$	$\|S \cap S'\|$	r_{bits}:true bits in $BF_\wedge$	Relative error			Probabilistic bounds
				Avg.	Max.	Std.dev.	
Overlap ratio varies							
800000	800000	50000	0.842	0.012	0.054	9.26E-3	**70%:**46635–53363 **90%:**45772–54225
800000	800000	100000	0.703	5.70E-3	0.023	4.51E-3	**70%:**96731–103266 **90%:**95892–104104
800000	800000	200000	0.474	2.57E-3	0.011	1.93E-3	**70%:**196915–203083 **90%:**196123–203874
800000	800000	400000	0.177	8.85E-4	4.52E-3	6.73E-4	**70%:**397245–402753 **90%:**396539–403458
Cardinality of of S' varies							
800000	100000	50000	0.327	2.72E-3	0.010	2 E-3	**70%:**49160–50839 **90%:**48945–51054
800000	200000	100000	0.306	1.86E-3	8.66E-3	1.42E-3	**70%:**98789–101209 **90%:**98480–101519
800000	400000	200000	0.264	1.34E-3	5.13E-3	1.02E-3	**70%:**198217–201782 **90%:**197760–202239
800000	800000	400000	0.177	8.85E-4	4.52E-3	6.73E-4	**70%:**397245–402753 **90%:**396539–403458

Table 5 Estimation accuracy for Bloom filter intersection - Influence of intersection characteristics.

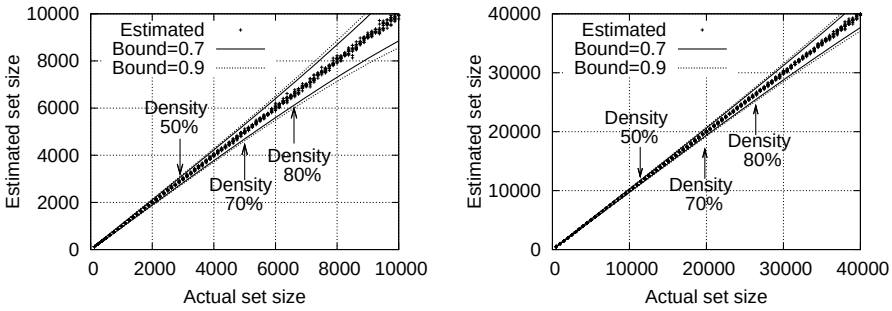

Fig. 10 Estimation accuracy for Block-partitioned Bloom filters: a. 4096 bits per block, and, b. 16384 bits per block, with 2 blocks and 1 hash function per block.

All blocks were initialized with a single hash function, as explained in Section 3. For each BBF configuration, we have set the maximum cardinality of S to the one that resulted to a BBF of density 0.9. To enable comparison of the results with the results for standard Bloom Filters, in the following experiments we have configured the BBFs such that their cumulative memory requirements corresponds to the memory requirements of the Bloom filters used in Section 5.4.1.

Influence of set cardinality. Figure 10 shows the relation between the estimated and actual set cardinality for two sample BBF configurations, with blocks of 4 Kbits and 16 Kbits. Table 6 presents further results for selected configurations.

We observe that cardinality estimation accuracy and tightness of probabilistic bounds are comparable to the corresponding accuracy and bounds for standard Bloom filters. In practice, the difference in estimation accuracy is negligible, i.e., breaking the Bloom filters into blocks does not negatively affect cardinality estimation accuracy. Recall from Section 4.1 that this is also the case with respect to false positive probability for membership tests.

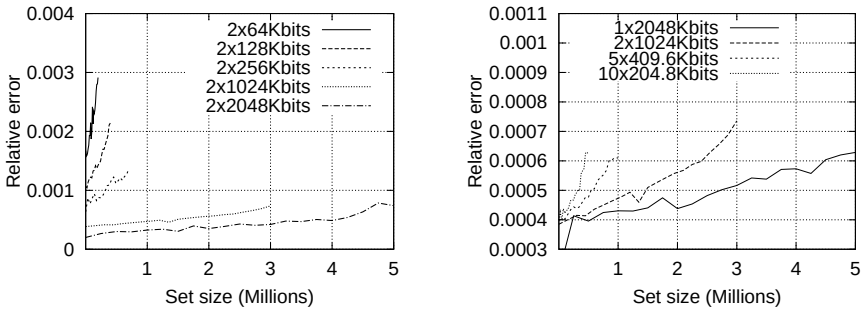

Fig. 11 Estimation accuracy for Block-partitioned Bloom filters: a. Influence o Block length, b. Influence of number of Blocks.

We also note that, similar to the case of standard Bloom filters, accuracy of cardinality estimation depends on the density of the BBF. For sparse BBFs, estimated and actual cardinalities differ only slightly; this difference increases for denser BBFs. Nevertheless, even for extremely dense BBFs, estimated cardinality still remains very close to the actual value. As expected, density also affects probabilistic bounds and standard deviation of relative error: for lower densities, probabilistic bounds are significantly tighter and standard deviation of relative error is also smaller.

Influence of block length. It is also interesting to see how block length affects cardinality estimation accuracy. Figure 11 a. plots relative error for Block-partitioned Bloom filters, constructed with various block lengths. The presented results are for BBFs with 2 blocks. We observe that increasing the block length for BBFs has a similar effect to increasing the filter length for standard Bloom filters (cf. Figure 6 a.). In particular, for a fixed set, cardinality estimations are more accurate at the BBFs with larger blocks. Furthermore, from the detailed results in Table 6 we can see that increasing the Block length has also a positive effect on probabilistic bounds and on standard deviation.

Finally, for a fixed density, relative error reduces when Block length increases (cf. Table 6). For example, for a density of 0.534, the BBF with two blocks of 128 Kbits has average relative error 1.21E-3, while the BBF with blocks of 256 Kbits has double the capacity for the same density, and a relative error of only 8.89E-4.

Influence of number of blocks. Figure 11 b. shows average relative error in relation to set cardinality for BBFs of different numbers of blocks. All presented results are for BBF structures with cumulative length of 2048 Kbits, i.e., the block length is selected such that the total structure length is 2048 Kbits. As expected, increasing the number of blocks has the same effect as increasing the number of hash functions in standard Bloom filters. Relative error increases with the number of blocks because length of each block is reduced linearly with the number of blocks, and the BBFs become overly dense sooner. Thereby, with respect to cardinality estimation, a smaller number of large blocks is preferable over more smaller blocks.

We also see that for BBF densities suitable for membership tests, the effect of number of blocks to cardinality estimation is negligible. For example, for 200 thousand elements, all BBFs in Figure 11 b. have a relative error of 0.00045 +/- 0.00005. Therefore, a system that requires both membership tests and cardinality estimation should configure the BBF for reducing the false positive probability for membership tests and for increasing the re-

Block length (Kbits)	Blocks	Items	Density	Relative error			Probabilistic bounds
				Avg.	Max.	Std.dev.	
Only number of elements varies							
4	2	3000	0.519	7.25E-3	0.030	5.47E-3	**70%**2837–3161 **90%**2794–3205
4	2	9000	0.889	9.19E-3	0.042	7.02E-3	**70%**8105–9893 **90%**7881–10117
Block length varies							
128	2	100000	0.534	1.21E-3	6.29E-3	9.24E-4	**70%**99043–100955 **90%**98764–101234
128	2	200000	0.783	1.42E-3	7.37E-3	1.1E-3	**70%**197523–202474 **90%**196808–203189
256	2	200000	0.534	8.89E-4	3.9E-3	6.92E-4	**70%**198648–201351 **90%**198251–201748
4096	2	1E7	0.908	1.19E-3	2.5E-3	3.93E-4	**70%**9964361–10035631 **90%**9953887–10046105
Number of blocks varies; the total BBF length remains constant							
2048	1	500000	0.212	3.68E-4	1.67E-3	2.8E-4	**70%:**497981–502018 **90%:**497385–502614
204.8	10	500000	0.908	6.4E-4	2.91E-3	4.64E-4	**70%:**496438–503559 **90%:**495404–504594

Table 6 Estimation accuracy for BBFs. Note that the cumulative length of the BBFs (i.e., Block length × number of blocks) corresponds to the Bloom filter length in the results of Table 3.

duction flexibility, according to the analysis presented in Section 3. By doing so, cardinality estimation will already be very accurate for most practical applications.

Summarizing, the experimental results confirm the suitability of Theorem 3 for cardinality estimation for Block-partitioned Bloom filters. For BBFs with moderate densities that are acceptable for membership tests, the theorem obtains accurate cardinality estimations and tight probabilistic bounds.

6 Applications

Block-partioned Bloom filters, as well as cardinality estimation, are useful for a wide range of applications. In this section we describe a few areas where current approaches can benefit from our contributions.

Distributed Query Processing. Cardinality estimation based on Bloom filters can provide the necessary statistics required for distributed query planning algorithms. In particular, most of the query planning algorithms for distributed databases share the same idea [29]: the query planner optimizes the execution order of the joins, by using selectivity estimates for each join to predict the network cost that each execution order would incur. Accurate selectivity estimates lead to plans which save significant network resources. However, computing the selectivity estimates requires extended network interaction between the query planner and the participating databases. By using Bloom filters and the proposed probabilistic cardinality estimation, the query planner can efficiently compute accurate selectivity estimates. For instance, to estimate the selectivity of an equi-join, the query planner can merge the Bloom filter representations of the join attributes with bitwise-AND, and use the resulting Bloom filter to estimate the cardinality of the join. The AND-merged Bloom filter can be further sent to the corresponding databases, for enabling distributed Bloom joins [2, 3].

Block-partitioned Bloom filters are useful for optimizing chains of Bloom joins (i.e., when more than two nodes participate). For chained Bloom joins, the query processing involves finding the intersection of n distributed sets $S_1, S_2, \ldots, S_n$, by using Bloom filters for reducing the network cost. The Bloom join algorithm proceeds as follows: (a) the Bloom filters $BF_1, BF_2, \ldots, BF_n$ of the remote sets are collected in the coordinator, i.e., the query initiator, (b) the Bloom filter of the intersection is estimated by joining the original Bloom filters with bitwise-AND, (c) the Bloom filter of the intersection is propagated back to the participating nodes for filtering out the elements that are not in the intersection, and, (d) the elements that appear to be in the Bloom filter intersection are transmitted back to the query initiator, where the actual intersection (join) is computed and the results are presented to the user. At step (c) of the above algorithm, it is beneficial for the coordinator to adjust the resolution of the intersection's Bloom filter to its density [6]. However, rebuilding a smaller Bloom filter from scratch is not possible, since the intersection is not yet materialized. If BBFs are used instead, the coordinator can dynamically and inexpensively reduce the length and minimize the required network resources.

Content Caching and Distribution Networks. Another application area for Block-partitioned Bloom filters includes distributed systems that currently exchange Bloom filter summaries of their contents over the network, for the purpose of distributed caching [14], or for optimizing content distribution [23, 30]. Depending on their capabilities, nodes in these systems may want to reduce the Bloom filter length before sending it over the network, for saving network resources. For instance, in Summary Cache [14], weak nodes or nodes under heavy load may choose to exchange smaller Bloom filters, for reducing their network cost in the expense of more false positives. The optimal Bloom filter length depends on the bandwidth of the sender and receiver, the cost of each false positive, the current network load and other network characteristics. Therefore, a Bloom filter may need to be reduced to many different lengths during its lifetime. Rebuilding the Bloom filter from scratch each time involves unnecessary delays and computational overhead, and requires keeping a copy of the set which may be impossible, e.g., in streaming data. With BBFs, this reduction can be performed dynamically and efficiently, and with a near optimal false positive probability.

P2P Systems. Block-partitioned and Dynamic Block-partitioned Bloom filters find a wide range of applications in P2P networks. Currently, several P2P systems employ standard Bloom filters as summaries for reducing the network costs. For example, Bloom filters are used for reducing the network resources [11, 15], and increasing the quality of the results [13]. All participating peers use Bloom filters of a fixed length, which is problematic in real-world P2P systems because some peers have significantly larger collections than others [31]. Also, peers with weaker network connections cannot reduce the length of their Bloom filters dynamically. BBFs and D-BBFs are a good replacement of standard Bloom filters for these systems, as they allow peers to dynamically adapt the Bloom filter length based on their collection size and on network characteristics.

Other P2P systems employ Attenuated Bloom filters (ABFs) for enabling query routing [32, 33]. Briefly, an ABF is an array of standard equi-length Bloom filters. Each peer constructs an ABF and uses it to summarize its contents (the first Bloom filter in the array), the contents of its immediate neighbors (the second Bloom filter), the contents of its second-order neighbors (the third Bloom filter), and so on. To limit the network requirements, the higher-order ABFs are constructed by merging the corresponding Bloom filters with bitwise OR. However, this merging sacrifices the mapping between elements and peers; it instead creates a mapping between elements and paths. Therefore, for query answering, the query needs to go through all the intermediary peers, using the inverse path of the Bloom filters. With BBFs we can avoid this issue, and still keep the network requirements upper bounded.

Each peer can reduce the resolution of the Bloom filters received from its neighbors, such that they all sum up to the same fixed cost. Since each Bloom filter will correspond to exactly one peer, the query can be routed directly to this peer.

Cardinality estimation for Bloom filters is also frequently required for P2P networks. For example, Bender et al. [13] but also Koloniari et al. [24] merge the Bloom filter summaries of peers with bitwise-AND, to detect the peers with the smaller and larger overlap. However, these works are based on the restricting assumption that all Bloom filters contain approximately the same number of elements. With our results, it is now possible to accurately estimate this overlap, even when this assumption is not true.

Streaming. Block-partitioned Bloom filters and Dynamic Block-partitioned Bloom filters are important for summarizing sets of unknown cardinalities, such as streams. The stream listeners are frequently unaware of the stream length as well as the number of distinct elements in the stream, since the stream is often generated dynamically. As such, they cannot initialize the Bloom filter properly. If they overestimate the stream length, they will generate a very large Bloom filter which cannot be easily sent over the network or stored. On the other hand, underestimating the stream length will lead to a very dense Bloom filter, with an increased false positive probability. Instead, the BBFs enable a dynamic reduction of the length with a near-optimal false positive probability, whereas the D-BBFs can also increase the capacity of the filter, whenever this is required.

Cardinality estimation can be applied in stream analysis applications, which frequently use Bloom filters for summarization. For these applications, estimating the number of distinct events which occur within a time period can now be performed without any additional effort. Similar requirements also occur frequently in the case of network routing, for improving the routing infrastructure [34], and in click stream analysis, e.g., [27].

7 Related work

Following the wide applicability of Bloom filters, literature is rich in extensions of the Bloom filter structure and its capabilities. The related work can be split in two categories: (a) extensions which allow Bloom filters to represent more complex information than just set membership, and, (b) extensions that optimize Bloom filters for specific contexts and applications.

Representing complex information. In [14], Fan et al. introduce Counting Bloom filters, which use a small counter at each position instead of just one bit. Counting Bloom filters server two purposes. First, they can be used to capture frequency statistics instead of just memberships. Second, they allow deletions of elements (by decrementing the respective counters). A limitation of Counting Bloom filters is that the counters are of fixed size, therefore they can be overflown if an element is very frequent. Spectral Bloom filters [20] address this limitation by using variable-length counters. They use an efficient indexing technique for the counters, which allows updates in constant time and enables better space usage compared to Counting Bloom filters. Kumar et al. [35] describe a similar probabilistic structure, called space-code Bloom filters. Finally, the Bloomier filters proposed by Chazelle et al. [19] make Bloom filters applicable more widely by enabling any kind of function to be represented by Bloom filters, not just set membership queries. Bloom histograms, introduced in [36], combine Bloom filters and histograms to build efficient indexes for path expression queries, e.g., for XPath query processing. They have been extended to multi-level Bloom histograms in [37]. Another approach to support path expression queries is [38].

It is straightforward to apply the principle of Block partitioning to these extended representations, to enable dynamic resolution reduction. With respect to cardinality estimation, an analysis in the line of the one presented here can be performed to cover counting, spectral, and space-code Bloom filters as well.

Bloom Filter Optimizations. Mitzenmacher in [18] proposed Compressed Bloom Filters. The approach employs the fact that sparse Bloom filters have a low entropy, therefore they can be compressed effectively. Therefore, instead of creating Bloom filters with the optimal density 0.5, the approach proposes creating larger but sparser Bloom filters, and compressing them to reduce their length. The resulting compressed Bloom filter maintains the same false positive probability as the original Bloom filter. Note however that this technique does not allow the user to choose the size that the Bloom filter will have after compression. Also, the compression cannot be performed on-the-fly on an existing Bloom filter. Instead, all the elements need to be re-hashed to a larger Bloom filter to enable compression.

Bloom filters are frequently used in networking hardware, e.g., routers or firewalls. Because computing and memory constraints for these systems differ significantly from software-level Bloom filters, specific optimizations are required [39–42]. Particularly interesting for our work is the structure of Aggregated Bloom Filters (*ABF*), proposed in [41]. Similar to Block-partitioned Bloom filters, ABFs split the bit set into several segments, one for each hash function. While ABFs resemble BBFs with respect to internal structure, their purpose is completely different. Instead of optimizing memory usage, ABFs are used to increase access performance for hardware implementations via parallelization. In contrast to BBFs, it is impossible to reduce the size of an ABF because the number of blocks is part of the hardware design, and cannot be reduced dynamically based on the observed Bloom filter density. The same applies to the work presented in [27].

Incremental Bloom Filters [43] were proposed for allowing the Bloom filters to extend, for accommodating increasing set cardinalities. Their functionality is very similar to Dynamic Bloom filters (cf. Section 4), but Incremental Bloom filters have advantages when the probability density function for the set cardinality is known. For Dynamic Block-partitioned Bloom filters, we have used Dynamic Bloom filters as a building block instead of Incremental Bloom Filters, because Incremental Bloom filters are more complex and do not offer any advantage in our context. Nevertheless, Dynamic Block-partitioned Bloom filters could as well be built over Incremental Bloom filters if it would be beneficial for the context.

Set Cardinality Estimation with other Data Structures. With respect to set cardinality estimation, we have proposed a technique which estimates the cardinality of a set from its Bloom filter representation. For completeness, we note that there are several other data structures which address the same problem, e.g., [44, 45]. Compared to these works, our approach focuses on scenarios where a Bloom filter would anyway be required for membership testing, or is already available, e.g., where Bloom joins are used. For these applications, our approach enables estimating the set cardinality with no additional cost. Furthermore, Bloom filters enable us to address scenarios which cannot be handled by previous works but are nevertheless valuable for distributed databases, e.g., finding the cardinality of the intersection of two sets.

8 Conclusions

Bloom filters are of paramount importance for distributed applications. They are used in many distributed settings, ranging from distributed databases to P2P networks and dis-

tributed collaborative systems. Particularly for distributed databases, they enable efficient distributed joins, and they are used to optimize standard as well as top-k queries.

In this work, we proposed two novel Bloom filter features which enable additional considerable optimizations for distributed databases. Our first contribution, the Block-partitioned Bloom filter, is a Bloom filter encoding which enables dynamic and near-optimal reduction of the filter's length and number of hash functions with practically no cost. As demonstrated, the reduction of length and number of hash functions enables significant saving of network resources in distributed query execution, and is directly applicable on existing distributed algorithms, e.g., Bloom joins. To the best of our knowledge, this is the first proposal of dynamic length reduction in Bloom filters. To enable extending the length of Bloom filters as well, we introduced Dynamic Block-partitioned Bloom filters. The new structure allows for both reduction and extension of the length of the Bloom filter, to account for the cardinality of the represented set and for application-specific requirements.

The second contribution of this work allows for cardinality estimation of Bloom filters, with strict probabilistic guarantees. Our analysis supports standard Bloom filters, Block-partitioned Bloom filters, and also Bloom filters of non-materialized sets, e.g., the union or intersection of two sets performed directly on their Bloom filter representations. Bloom filter cardinality estimation is important for effective query planning in distributed databases, e.g., for estimating the selectivity of equi-joins. In this work, we already identified several algorithms in distributed databases and related areas which immediately benefit from the Bloom filter cardinality estimation approach, without any additional cost.

For both contributions we provided a comprehensive theoretical analysis as well as a large-scale experimental evaluation, covering a wide range of application scenarios. Both theoretical and experimental results confirm the general applicability of our work to many different applications and settings.

An interesting direction for future research is extending the aforementioned contributions to derived Bloom filter variants, such as Counting Bloom filters and Spectral Bloom filters. For the case of BBFs and D-BBFs, their variants using Counting and Spectral Bloom filters would enable a compact representation of multisets with dynamic resolution reduction, useful in many distributed applications. For the case of cardinality estimation, such extensions would unfold new application areas, where the number of instances in multisets is also important, and not only the number of distinct elements.

References

1. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *ACM Communications*, vol. 13, no. 7, pp. 422–426, 1970.
2. K. Bratbergsengen, "Hashing methods and relational algebra operations." in *Proceedings of the Tenth International Conference on Very Large Data Bases (VLDB)*, 1984, pp. 323–333.
3. L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries." in *Proceedings of the Twelfth International Conference on Very Large Data Bases (VLDB)*, 1986, pp. 149–159.
4. J. K. Mullin, "Optimal semijoins for distributed database systems." *IEEE Transactions of Software Engineerings*, vol. 16, no. 5, pp. 558–560, 1990.
5. L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski, "Improving distributed join efficiency with extended bloom filter operations," in *Proceedings of 21st International Conference on Advanced Information Networking and Applications (AINA)*, 2007, pp. 187–194.
6. S. Ramesh, O. Papapetrou, and W. Siberski, "Optimizing distributed joins with bloom filters," in *Proceedings of International Conference of Distributed Computing and Internet Technology (ICDCIT)*, 2008.

7. S. Michel, P. Triantafillou, and G. Weikum, "Klee: a framework for distributed top-k query algorithms," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005, pp. 637–648.
8. T. Neumann, M. Bender, S. Michel, R. Schenkel, P. Triantafillou, and G. Weikum, "Distributed top-k aggregation queries at large," *Distributed and Parallel Databases*, vol. 26, no. 1, pp. 3–27, 2009.
9. G. Koloniari and E. Pitoura, "Content-based routing of path queries in peer-to-peer systems," in *Proceedings of International Conference on Extending Database Technology (EDBT)*, 2004, pp. 29–47.
10. A. Kumar, J. J. Xu, and E. W. Zegura, "Efficient and scalable query routing for unstructured peer-to-peer networks," in *Proceedings of the 24rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2005.
11. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen, "PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," in *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.
12. P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 21–40.
13. M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer, "Improving collection selection with overlap awareness in p2p search engines." in *Proceedings of the 28th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, 2005, pp. 67–74.
14. L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.
15. J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 767–780, 2004.
16. N. Anciaux, M. Benzine, L. Bouganim, P. Pucheral, and D. Shasha, "Revelation on demand," *Distributed and Parallel Databases*, vol. 25, no. 1-2, pp. 5–28, 2009.
17. D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2006.
18. M. Mitzenmacher, "Compressed bloom filters." *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, 2002.
19. B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, 2004, pp. 30–39.
20. S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 241–252.
21. R. Zhou, K. Hwang, and M. Cai, "Gossiptrust for fast reputation aggregation in peer-to-peer networks," *IEEE Trans. on Knowl. and Data Eng.*, vol. 20, no. 9, pp. 1282–1295, 2008.
22. D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *Transactions on Knowledge and Data Engineering (TKDE)*, vol. 22, no. 1, 2010.
23. J. Yan and P. L. Cho, "Enhancing collaborative spam detection with bloom filters," *Computer Security Applications Conference, Annual*, vol. 0, pp. 414–428, 2006.
24. G. Koloniari, Y. Petrakis, and E. Pitoura, "Content-based overlay networks of xml peers based on multi-level bloom filters," in *Proceedings of VLDB International Workshop on Databases, Information Systems and Peer-to-Peer Computing*. Springer-Verlag, 2003, pp. 232–247.
25. A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Allerton Conference*, 2002.
26. A. C. Snoeren, "Hash-based ip traceback," in *SIGCOMM*, 2001, pp. 3–14.
27. A. Metwally, D. Agrawal, and A. El Abbadi, "Duplicate detection in click streams," in *Proceedings of the 14th International Conference on World Wide Web (WWW'05)*. New York, NY, USA: ACM, 2005, pp. 12–21.
28. R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, 2000.
29. D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys*, vol. 32, no. 4, pp. 422–469, 2000.
30. A. L. Chervenak, E. Deelman, I. T. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Z. Kunszt, M. Ripeanu, R. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: a framework for constructing scalable replica location services," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1–17.
31. S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, 2002.

32. J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Y. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, USA, 2000, pp. 190–201.
33. S. C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *INFOCOM*, 2002.
34. S. Muthukrishnan, "Data streams: Algorithms and applications," *Foundations & Trends in Theoretical Computer*, vol. 1, no. 2, 2005.
35. A. Kumar, J. Xu, J. Wang, O. Spatscheck, and L. Li, "Space-code bloom filter for efficient per-flow traffic measurement," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2004.
36. W. Wang, H. Jiang, H. Lu, and J. X. Yu, "Bloom histogram: Path selectivity estimation for xml data with updates," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 240–251.
37. G. Koloniari and E. Pitoura, "Distributed structural relaxation of xpath queries," in *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, 2009, pp. 529–540.
38. S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun, "Xml processing in dht networks," in *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, 2008, pp. 606–615.
39. H. Song, T. S. Sproull, M. Attig, and J. W. Lockwood, "Snort offloader: A reconfigurable hardware NIDS filter," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 493–498.
40. O. Erdogan and P. Cao, "Hash-av: fast virus signature scanning by cache-resident filters," *International Journal of Security and Networks*, vol. 2, no. 1/2, pp. 50–59, 2007.
41. N. S. Artan, K. Sinkar, J. Patel, and H. J. Chao, "Aggregated bloom filters for intrusion detection and prevention hardware," in *Proceedings of the Global Communications Conference (GLOBECOM)*, 2007, pp. 349–354.
42. E. Safi, A. Moshovos, and A. G. Veneris, "L-CBF: A low-power, fast counting bloom filter architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 6, pp. 628–638, 2008.
43. F. Hao, M. S. Kodialam, and T. V. Lakshman, "Incremental bloom filters," in *Proceedings of the 27th IEEE International Conference on Computer Communications (INFOCOM)*, 2008, pp. 1067–1075.
44. P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 182–209, 1985.
45. Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM'02)*, 2002, pp. 1–10.