Improving distributed join efficiency with extended bloom filter operations

Loizos Michael*, Wolfgang Nejdl◇, Odysseas Papapetrou◇, Wolf Siberski◇
*Division of Engineering and Applied Sciences, Harvard University loizos@eecs.harvard.edu
◇L3S Research Center, Leibniz Universität Hannover {nejdl,papapetrou,siberski}@l3s.de

Abstract

Bloom filter based algorithms have proven successful as very efficient technique to reduce communication costs of database joins in a distributed setting. However, the full potential of bloom filters has not yet been exploited. Especially in the case of multi-joins, where the data is distributed among several sites, additional optimization opportunities arise, which require new bloom filter operations and computations. In this paper, we present these extensions and point out how they improve the performance of such distributed joins. While the paper focuses on efficient join computation, the described extensions are applicable to a wide range of usages, where bloom filters are facilitated for compressed set representation.

1 Introduction

In a distributed database setting, joins are expensive operations, especially with respect to communication costs. Assume that we want to compute $S \bowtie T$ at the site holding T, called master site. Basic join algorithms require that all tuples in S (or at least a vertical subset of them) are sent to the master site, where the actual join computation, - the intersection between T and S based on the join condition - takes place.

Instead of sending the actual data, it is sufficient to send a compressed form of the set of tuples forming S, with just enough information to test set membership. For this, bloom filters are the ideal choice.

Bloom filters The Bloom filter data structure was proposed in [4], as a space-efficient representation of sets $S = \{e_1, e_2, e_3 \dots e_n\}$ of n elements from a universe U. A bloom filter consists of an array of m bits and a set of k independent hash functions $F = \{f_1, f_2 \dots f_k\}$, which hash elements of U to an integer in the range of $[1, m]$. The m bits are initially set to 0 in an empty bloom filter[1]. An element e is inserted into the bloom filter by setting all positions $f_i(e)$ of the bit array to 1.

Bloom filters allow membership queries without the need of the original collection. For any given element $e \in U$, we can conclude that e is not present in the original collection if at least one of the positions computed by the hash functions of the bloom filter points to a bit which is set to 0. However, bloom filters allow 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 that such a membership test yields a false positive is $P(false-positive) \approx (1-e^{-kn/m})^k$. The information density of a bit filter is optimal when the probability of each bit to be set is $1/2$. For a bloom filter, this is the case when setting the number of hash functions to $k \approx \frac{m}{n} * \ln(2)$.

Bloom filters have gained a wide spectrum of applications, including cache management [11], routing in peer-to-peer systems [14], novelty estimation in P2P [2], and queue management [7]. A recent survey can be found in [6].

Since their invention, several extensions to bloom filters have been proposed. Mitzenmacher [19] shows how to compress bloom filters optimally. Chazelle et al. [8] propose the *bloomier filters* which enable any kind of function to be represented with bloom filters, not only the membership function. Fan et al. [11] introduced *counting bloom filters*, which allow to manage insertions *and* deletions of elements. *Spectral bloom filters* take this one step further and introduce variable-length counters, allowing arbitrarily large counters for set elements [10]. Another approach to multiset representation, the *space-code bloom filter* is described in [13]. Finally, [12] presents *dynamic bloom filters*, which dynamically adapt their size to the number of inserted elements.

Bloom filter based joins The first hash-based join algorithm has been described in [1]. The simple hash-join works as follows: Suppose we want to compute $S \bowtie T$, where S is the smaller relation. S is called the *building* relation, because all tuples from S are added to a main memory

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

hash table[2]. Then, each tuple from T (the *probing* relation) is accessed and used to probe this hash table. If probing succeeds, a new result tuple is created. In most cases the hashing join algorithms perform better than other join algorithms such as the sort-merge join [21, 17].

To reduce communication costs in a distributed setting, a semi-join stage can be applied before the actual join [3]. Suppose $site_S$ is holding S, $site_T$ holds T, and the join condition is $S.a = T.b$. Then, $site_S$ first sends just $\pi_a(S)$ to $site_T$. $site_T$ sends back all tuples from T for which probing succeeded to $site_S$, where the final join is computed. If the join condition is highly selective, this can save significant communication costs, because only small fractions of tuples of S need to be transmitted.

The *Bloomjoin* algorithm [5, 18] reduces the amount of data transmitted further by encoding $\pi_a(S)$ in a bloom filter. It proceeds as follows: First $site_S$ produces a bloom filter BF_S, including the join key $S.a$ for all its records. The bloom filter BF_S is then sent to $site_T$, and used for filtering the records of T that do not satisfy the join, i.e., $T.b$ is not included in BF_S. The rest of the records are then sent to $site_S$, where the actual join can occur, and any false positives can be filtered. [18] shows that Bloomjoin consistently outperforms the basic semi-join algorithm.

Mullin [20] points out that this two-stage join only saves costs when the filtering at $site_T$ does filter a significant amount of tuples, which is not always the case. He proposes to extend the approach to a multi-stage process, where the sites start with a very small bloom filter, and increase its size until the additional costs for transmitting the bloom filter outweigh the gains.

Another approach includes Positionally Encoded Record Filters (PERFjoin) [15]. Positionally Encoded Record Filters (PERFs) are bitvectors, encoding the matching records in the backward direction of the 2-way semi-joins. Namely, executing the $R \bowtie S$ as $S \ltimes R$ and following $R \ltimes S$ can be optimized in the following manner:(a) S projects and sends the join attributes to R, (b) R uses the order proposed from S for constructing a bitvector, where it sets the bits to 1 if the respective record in S should be included in the relation. The original approach can also be used as an extension to the bloom join technique, for eliminating all false positives. The authors show clear network gains over naively implemented two-way semijoins that transmit the whole relation instead of only the join attributes. The approach however is beneficiary only in cases of very low selectivity, because traditional compression proposed in the paper has poorer performance than the original bloom filter technique in the other cases. Also it cannot give benefit in cases where the join attributes are bigger than the integer length (i.e. strings).

Based on these techniques, algorithms to optimize multi-join query plans have been devised. [9] shows how to combine joins and semi-joins to minimize network transmission costs of the distributed execution. [23] presents an algorithm for minimizing response times. [16] focuses on optimizing the total processing cost of all sub-queries at the database nodes.

These works on query planning for distributed joins take bloom filter-based transmission for granted. However, a closer look shows that it is possible to improve the query execution efficiency further by optimizing the bloom filters exchanged between sites. This is especially true if sites cache bloom filters used for distributed joins. We present such optimization opportunities in three areas:

- While the optimal bloom filter size has been determined for a two-site join [20], the situation becomes more complex when bloom filters from several sites need to be combined at one master site. We show how to pre-compute error probabilities for composed bloom filters, and point out how this affects the optimal bloom filter size.

- When pipelining the bloom filter based set intersection, in many cases the number of items in the set decreases significantly. We show how bloom filter size can be reduced efficiently without re-hashing; this allows intermediate sites to optimize the size of the bloom filter they forward to the next site in the pipeline.

- To determine the optimal order of joins, the master site needs to know (or to estimate) the join selectivity at each site. While it is possible to ask each site for this data, a master site can estimate it efficiently if the respective bloom filters for the join condition are already available. We show how to estimate the size of a set from the number of '1' bits in the corresponding bloom filter. This allows estimating the selectivity of a join based on the conjunction of the respective bloom filters.

It is interesting to note that these optimizations can be applied directly to any distributed querying algorithm based on semi-joins.

In the following, we consider only natural joins, i.e., join conditions of the form $S.a = T.a$. The extension to arbitrary equality conditions is straightforward. Joins with non-equality conditions cannot be optimized using bloom filters, and therefore are not taken into consideration.

2 Bloom Filter Composition Operations

Motivation Suppose we have sites $site_{P1} \cdots site_{Pm}$ collecting person information (in a table PERSON(ID, NAME,

[2] we use the terminology of [22].

$\cdots$), and other sites $site_{Q1} \cdots site_{Qn}$ collecting publication information (in tables PUBLICATION(ID, TITLE, DATE, $\cdots$), AUTHOR(PERS_ID, PUB_ID). A typical query is to select publications of authors with a specific name:
$(\bigcup_{i=1\cdots m} \sigma_{\text{NAME='Foo'}}(\text{PERSON}_i)) \bowtie_{\text{PERSON.ID=AUTHOR.PERS\_ID}}$
$(\bigcup_{i=1\cdots n} \text{AUTHOR}_i) \bowtie_{\text{AUTHOR.PUB\_ID=PUBLICATION.ID}}$
$(\bigcup_{i=1\cdots n} \text{PUBLICATION}_i)$

Suppose, bloom filters $BF_{\pi_{PERS\_ID}(AUTHOR_i)}$ are cached at the master site. Then, an efficient distributed query plan would execute the query as follows:

1. Retrieve bloom filters $BF_{\pi_{ID}(\sigma_{NAME='Foo'}(PERSON_i))}$ from sites $site_{Pi}$
2. Compute the union of these bloom filters by bitwise OR to $BF_{\cup PERSON}$
3. Compute the union of $AUTHOR_i.PERS\_ID$ by bitwise OR of the respective cached bloom filters to $BF_{\cup AUTHOR}$
4. Compute the PERSON-AUTHOR semi-join by bitwise AND of $BF_{\cup PERSON}$ and $BF_{\cup AUTHOR}$ to $BF_{(\cup PERSON)\ltimes(\cup AUTHOR)}$
5. Send the resulting bloom filter to sites $site_{Qi}$ to compute the join on PUBLICATION.
6. Collect the results, check for false positives, and merge result tuples.

In general, we need bitwise AND to compute the intersections required by joins, and bitwise OR to compute unions in case of horizontal fragmentation. In both cases, bloom filters facilitate the required unions and joins with reduced network overhead. This applies even more when bloom filters for foreign key attributes are cached at the coordinating site. However, we need to take care that the false-positive error rate is strictly controlled. This can only be done by estimating error rates for the mentioned composition operations, and requesting (rsp. caching) bloom filters of the appropriate size from participating sites.

In the following, we analyze the error probabilities for computing set intersection and union by bitwise AND rsp. OR of the involved bloom filters. The operators, while formally defined for only two bloom filters, are trivially extended for an arbitrary number of parameters using recursion: $op(BF_1, BF_2, \ldots BF_n) = op(BF_1, op(BF_2, \ldots (op(BF_{n-1}, BF_n))))$.

Set union with bloom filters Computing the union of two sets A and B based on their bloom filters BF_A rsp. BF_B is only possible if BF_A and BF_B have the same size and share the same hash functions. In this case, $BF_{A\cup B} = BF_A \wedge BF_B$, where $\wedge$ denotes bitwise OR.

Let us denote the error probability on bloom filter BF_1 as P_{error_1} and the error probability on bloom filter BF_2 as P_{error_2}. P_{error} is calculated using the probability of any random bit of the bloom filter to be set to 1. This probability $P(bit\ set\ to\ 1)$ is calculated based on the number of the records already hashed to the bloom filter $P(bit\ set\ to\ 1) = 1 - \left(1 - \frac{1}{m}\right)^{kn}$. Then, P_{error} is the probability that all k bits for a random record are set to 1, which equals to $P_{error} = P(bit\ set\ to\ 1)^k$.

So, $P_{error_1} = \left(1 - \left(1 - \frac{1}{m_1}\right)^{k_1 n_1}\right)^{k_1}$ and $P_{error_2} = \left(1 - \left(1 - \frac{1}{m_2}\right)^{k_2 n_2}\right)^{k_2}$ and, as a prerequisite $k_1 = k_2 = k$ and $m_1 = m_2 = m$. We could try to similarly calculate $P_{error_{OR}} = \left(1 - \left(1 - \frac{1}{m}\right)^{k n_{OR}}\right)^k$, where n_{OR} is the number of records contained in $BF_{OR(BF_1,BF_2)}$. However, just adding the number of records from BF_1 and BF_2 is not accurate, since some records may exist in both the tables. So, $P_{error_{OR}} <= \left(1 - \left(1 - \frac{1}{m}\right)^{k(n_1+n_2)}\right)^k$ can serve as an upper bound for the error probability $P_{error_{OR}}$.

A lower bound can also be derived for $P_{error_{OR}}$: $max(n_1, n_2) \leq n_{OR} \Rightarrow max(P_{error_1}, P_{error_2}) \leq P_{error_{OR}}$. Thus, the error probability bounds are

$$max(P_{error_1}, P_{error_2}) \leq P_{error_{OR}} \leq \left(1 - \left(1 - \frac{1}{m}\right)^{k(n_1+n_2)}\right)^k$$

If the distribution of the objects in the represented sets is independent – the most frequent case –, we can compute a better approximation by regarding the error probability on bit-level. For a bit to be set in $BF_{OR(BF_1,BF_2)}$, it has to be set either in BF_1 or BF_2 (or both). Therefore, the probability $P(bit\ set\ to\ 1, BF_{OR(BF_1,BF_2)}) = P(bit\ set\ to\ 1, BF_1) + P(bit\ set\ to\ 1, BF_2) - P(bit\ set\ to\ 1, BF_1) * P(bit\ set\ to\ 1, BF_2)$. The probability of a bit to be true in BF_1 is $P(bit\ set\ to\ 1, BF_1) = 1 - \left(1 - \frac{1}{m}\right)^{kn_1}$. Similarly, the probability for the same bit for BF_2 is $P(bit\ set\ to\ 1, BF_2) = 1 - \left(1 - \frac{1}{m}\right)^{kn_2}$. This gives us a probability

$$P(bit\ set\ to\ 1, BF_{OR}) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn_1}\right) + \left(1 - \left(1 - \frac{1}{m}\right)^{kn_2}\right) - \left(1 - \left(1 - \frac{1}{m}\right)^{kn_1}\right) * \left(1 - \left(1 - \frac{1}{m}\right)^{kn_2}\right)$$

Consequently, the error probability for $BF_{OR(BF_1,BF_2)}$ is

$$P_{error}(BF_{OR(BF_1,BF_2)}) = \left(\left(1-\left(1-\frac{1}{m}\right)^{kn_1}\right) + \left(1-\left(1-\frac{1}{m}\right)^{kn_2}\right) - \left(1-\left(1-\frac{1}{m}\right)^{kn_1}\right) * \left(1-\left(1-\frac{1}{m}\right)^{kn_2}\right)\right)^k$$

This probability is less than $P_{error_1} + P_{error_2}$ and can be approximated as

$$P_{error_{OR}} \approx Max(2 * P_{error_1} - P^2_{error_1}, 2 * P_{error_2} - P^2_{error_2})$$

Set intersection with bloom filters As with set union, we assume that the involved bloom filters have the same size and share their hash functions. Let n_{AND} be the number of the objects in the set intersection. Then: $n_{AND} \leq Min(n_1, n_2) \Rightarrow P_{error_{AND}} \leq Min(P_{error_1}, P_{error_2})$ (this follows directly from the formula of P_{error}).

Again, for an independent object distribution a better approximation is possible. In this case, the probability of a bit to be true in the combined bloom filter is $P(bit\ set\ to\ 1, BF_{AND(BF_1,BF_2)}) = P(bit\ set\ to\ 1, BF_1) * P(bit\ set\ to\ 1, BF_2) = \left(\left(1-\left(1-\frac{1}{m}\right)^{kn_1}\right) * \left(1-\left(1-\frac{1}{m}\right)^{kn_2}\right)\right)^k$. This probability is significantly less than P_{error_1} and P_{error_2}.

3 Reduction of Bloom Filter Resolution

Motivation Suppose we want to compute $R_1 \bowtie_{R_1.a=R_2.a} R_2 \bowtie_{R_2.a=R_3.a} R_3 \bowtie_{R_3.a=R_4.a} R_4$. If the relations R_i are large and the respective intersections are significantly smaller than the relation sizes, it is beneficial to avoid the sending of complete bloom filters $BF_{R_i.a}$ to a coordinating site. In that case, we would choose pipelined computation, and after each semi-join reduce the resulting bloom filter to its optimal size before forwarding it:

1. compute $BF_{R_1.a}$ at $site_1$ (or use cached BF) and send it to $site_2$
2. $site_2$ uses cached $BF_{R_2.a}$, adapts it to the size of $BF_{R_1.a}$ and computes $BF_{R_1 \ltimes R_2} = BF_{R_1.a} \wedge BF_{R_2.a}$
3. $site_2$ reduces size of $BF_{R_1 \ltimes R_2}$ to optimum and sends it to $site_3$
4. repeat semi-join computation until $BF_{R_1 \ltimes R_2 \ltimes R_3 \ltimes R_4}$ is produced at $site_4$
5. send back matching tuples of R_4 to $site_3$
6. compute $R_3 \bowtie R_4$ and send it to $site_2$
7. repeat join computation until $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ is produced at $site_1$

While we could re-create a new bloom filter of optimal size at each site, this is much more expensive than just computing bitwise AND of the cached bloom filters, and then reducing the bit array size without rehashing. For the latter, we need to (a) calculate the size satisfying our required error probability, and (b) efficiently shrink the bit array.

We now describe our approach for reducing the resolution of a bloom filter in the absence of the original collection. For comparison purposes, we first describe the naive approach based on mapping the large bloom filter to a smaller one (i.e. using the modulo operation) and show why this is not efficient.

Naive resolution reduction Assume we want to reduce a large bloom filter $BF$of length l to a smaller BF' of length l', where l/l' is an integer. A simple mapping function like $f(x) = (x \bmod l')$ can be used to map the bloom filter values as well as the bloom filter functions to the smaller bloom filter. BF' is initialized with no bit set, and for each bit $BF[b]$ in the original bloom filter, if $BF[b]$ is set, we set $BF'[b \bmod l']$ in the reduced bloom filter.

For this approach, the new error probability is much higher than the optimal probability for the same collection represented by a bloom filter of the same length l'. In particular, assuming a uniform hashing, BF' has an increased error probability of $P'(false\ positive) \approx (1 - e^{-kn/S})^k$. Figure 1 shows the error probability for naive size reduction and the optimal error probability for a sample bloom filter of size 32768, containing 400 objects. When this large bloom filter is reduced to 0.125 of the original (i.e., to 4096 bits) the error probability caused by naive reduction is already 0.804, while the optimal error probability would have been 0.007.

The problem is that the number of the hash functions is not reduced with the size of the bloom filter. We know that to achieve the optimal error probability, the number of hash functions has to be chosen such that the density of the bloom filter becomes 0.5. The reduction of such a bloom filter to half its size already results in a density of approximately 0.75. As the error probability grows exponentially with bloom filter density, the naive approach leads quickly to very high error probabilities. We show how to avoid this in the next section

Block-partitioned bloom filters The key to reduced error probabilities is to allow adapting the number of hash functions without the need to rehash all objects. This can be achieved by composing the bloom filter from small, independent bloom filter blocks. Each of these blocks contains all objects, hashed with different functions. Suppose a block-partitioned bloom filter of size l_{max}, reducible to a minimum size of l_{min}, is required. As first and second blocks, bloom filters of size l_{min}, each with the optimized

number of different hash functions are created, populated with all objects, and concatenated. As third block, a bloom filter of size $2 \cdot l_{min}$ is populated, and appended. The process continues until the size of the l_{max} is reached.

The reduction step for block-partitioned bloom filters is trivial: to reduce the resolution of a given filter with Φ hash functions to a size of 2^{ψ}, we just take the ψ first bits and the $\frac{\Phi}{2^{\chi-\psi}}$ first hash functions as new bloom filter.

Note that the approach is not limited to filter sizes of 2^n. If other sizes are required, they can easily be formed by concatenating block-partitioned bloom filters of different sizes. In this case, the reduction step needs to extract the respective reduced blocks from each of these filters, and re-concatenate the extracted blocks.

While on the surface there seems to be a similarity between block-partitioned bloom filters and dynamic bloom filters [12], the approaches are actually different. The purpose of dynamic bloom filters is to grow, as more and more objects are added. However, dynamic bloom filters can't be efficiently reduced. Block-partitioned bloom filters exhibit exactly the opposite characteristic: they can be reduced easily, but it is not possible to add new blocks, because any block needs to contain all objects.

Analysis: We sketch the analysis for finding the error probability for block-partitioned bloom filters. The analysis is restricted on filters of length l_max, where $log_2(l_max)$ is an integer. As noted above, this poses no restriction on the actual size.

The false positive error probability can be calculated recursively. A false-positive error occurs on a block-partitioned bloom filter of length l_{max} when both the last block and the remaining filter (each of length $l_{max}/2$) return a false positive error. The solution of the recursive equation $P(false\ positive, length = l_{max}) = P(false\ positive, length = \frac{l_{max}}{2})^2$ is $P(false\ positive, length = l_{max}) = P(false\ positive, length = l_{min})^{l_{max}/l_{min}}$, where l_{min} is the length of the smallest block, the basic building block of the full bloom filter. Note that there is a lower limit for l_{min}, the smallest block size; is it is too small, in the worst case all its bits will be set to one, rendering it useless. Therefore, l_{min} has to be chosen to satisfy the condition $\left(1 - \left(1 - \frac{1}{l_{min}}\right)^{kn}\right)^k \ll 1$.

Comparison: Figure 1 shows the false positive error probabilities for naively reduced bloom filters and block-partitioned ones, compared to the optimal size (if the objects would be re-hashed). The graph is based on a sample bloom filter of length $l_{max} = 32768$, containing 400 objects. The initial number of hash functions is 57, the optimal number for this setting. For practical issues the only the error rates for reduction sizes between 512 and 8192 bits are shown.

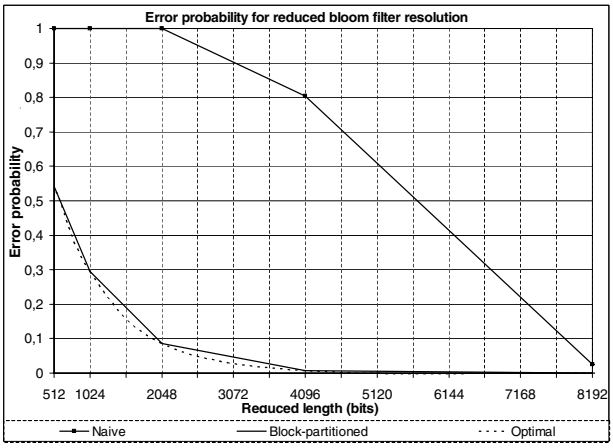

Figure 1. False positive error probabilities with different bloom filter reduction techniques

For larger reduction sizes, the difference between both approaches is not significant. The figure shows that naive reduction causes extreme high error probabilities. On the other hand, we see that block-partitioned bloom filters exhibit a near-optimal error probability for all reduction sizes, without requiring a rehashing of the original collection.

4 Estimation of Bloom Filter Set Size

Motivation Consider the case of the following query: $Q_1 : R_1 \bowtie_{R_1.a=R_2.a} R_2 \bowtie_{R_2.a=R_3.a} R_3 \bowtie_{R_3.b=R_4.b} R_4 \bowtie_{R_4.b=R_5.b} R_5$. In contrast to the previous scenarios, the joins occurring in this query are based on more than one attribute (R_1 to R_3 are joined on a, R_3 to R_5 are joined on b etc.). For this type of queries, bloom filters alone are not sufficient for filtering all non-satisfying records. Instead, any evaluation of such queries has to partially execute all sub-queries, then exchange a possibly large amount of possibly non-satisfying records, and finally filter them (progressively or centrally in a moderator). In our example, a possible query plan for Q_1 is:

- Break the query into sub-queries $SQ_1, SQ_2 \ldots SQ_m$ such that each sub-query has the maximum length of joins with the same attribute. For Q_1, that would be $SQ_1 = R_1 \bowtie_{R_1.a=R_2.a} R_2 \bowtie_{R_2.a=R_3.a} R_3$ and $SQ_1 = R_3 \bowtie_{R_3.b=R_4.b} R_4 \bowtie_{R_4.b=R_5.b} R_5$.
- Use cached bloom filters on the join attributes to compute BF_{AND_i} bloom filters of each sub-query SQ_i, e.g., $BF_{AND_i} = BF_{R_1.a} \wedge BF_{R_2.a} \wedge BF_{R_3.a}$ for sub-query SQ_1.
- Attach $BF_{AND_1}, BF_{AND_2} \ldots BF_{AND_m}$ to the sub-queries, and use pipeline computation to execute them.

The above plans, although requiring significantly less network transmissions than simple joins, can still be sub-optimal. To increase efficiency as well as reduce transmission costs, we have to order the sub-query evaluation in the pipeline so that the order of the sequential execution of the joins reduces the size of intermediate result sets as early as possible. For instance, if SQ_1 in the above example would result in only 1 record, it would be wise to first execute SQ_1, and then pipeline the 1 result record to the sub-query processing. However, asking each of the sites for a count on the number of local records that satisfy the whole query is not feasible. Due to the different join attributes, the sites cannot locally compute the number of records remaining after the joins. Estimating the cardinality of each sub-query based on the density of the joined bloom filters does not work either, since each of the sub-queries may produce different bloom filters sizes (i.e. by combining the optimizations already proposed from section 3).

Therefore, we devised a formula to derive the estimated object set size from a given bloom filter. Now, a moderator, i.e. the query initiator, can first execute the semi-joins for sub-queries SQ_i, and then use this formula to estimate join selectivities from the resulting bloom filters BF_{AND_i}. The joins are then ordered and executed based on these estimations, analogous to the selectivity statistics used in centralized databases.

Set size estimation An estimation of elements hashed into a bloom filter can be derived based on the bits set to true in the bloom filter as follows.

Lemma 4.1 *The expected number of true bits in a bloom filter of length m with k hash functions after n elements were hashed is:* $\hat{S}(n) = m * \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$. *Also, the following inequalities hold:*

Upper bound: *The probability of the number of true bits to be more than* $(1+\delta) * \hat{S}(n)$ *is* $P(\#\ true\ bits > (1+\delta) * \hat{S}(n) | n) \leq e^{-\hat{S}(n)\delta^2/3}$.

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

Proof Given a bloom filter of size m with k hash functions and n elements hashed into it, we can compute the expected number of true bits as follows. For this task, we define the 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 $\hat{S}(n) = \sum_{i=1}^{m} P(i = true) = m * \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$. 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.

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

$$
\begin{aligned}
P(i = true) = \frac{t}{m} = 1 - \left(1 - \frac{1}{m}\right)^{k\hat{S}^{-1}(t)} &\Rightarrow \\
\left(1 - \frac{1}{m}\right)^{k\hat{S}^{-1}(t)} = 1 - \frac{t}{m} &\Rightarrow \\
k * \hat{S}^{-1}(t) * ln\left(1 - \frac{1}{m}\right) = ln\left(1 - \frac{t}{m}\right) &\Rightarrow \\
\hat{S}^{-1}(t) = \frac{ln\left(1 - \frac{t}{m}\right)}{k * ln\left(1 - \frac{1}{m}\right)} &
\end{aligned}
$$

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

Theorem 4.2 *Given a bloom filter BF of length m with k hash functions and t bits set to true. For any* n_l, n_r *such that* $\hat{S}^{-1}(\frac{t-1}{2}) \leq n_l \leq \hat{S}^{-1}(t-1)$ *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* $1 - e^{-\frac{(t-1-\hat{S}(n_l))^2}{3\hat{S}(n_l)}} - e^{-\frac{(t+1-\hat{S}(n_r))^2}{2\hat{S}(n_r)}}$.

Proof If the number of documents 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 $(1+\delta_l) * \hat{S}(n_l) < t$, we obtain by lemma 4.1 that this probability is $P(\#\ true\ bits > (1+\delta_l) * \hat{S}(n_l) | n_l) \leq e^{-\hat{S}(n_l)\delta_l^2/3}$. Similarly, if the number of documents 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 $(1-\delta_r) * \hat{S}(n_r) > t$, we obtain by lemma 4.1 that this probability is $P(\#\ true\ bits < (1-\delta_r) * \hat{S}(n_r) | n_r) \leq e^{-\hat{S}(n_r)\delta_r^2/2}$.

For choices of $n_l, \delta_l, n_r, \delta_r$ as described above, we get that with probability $1 - e^{-\hat{S}(n_l)\delta_l^2/3} - e^{-\hat{S}(n_r)\delta_r^2/2}$, the number of documents is in the range (n_l, n_r). We compute a value for δ_l such that $(1+\delta_l) * \hat{S}(n_l) < t$. 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 $(1-\delta_r) * \hat{S}(n_r) > t$. Clearly, $\delta_r = \frac{\hat{S}(n_r)-t-1}{\hat{S}(n_r)}$ satisfies the inequality.

We conclude that with probability $1 - e^{-\frac{(t-1-\hat{S}(n_l))^2}{3\hat{S}(n_l)}} - e^{-\frac{(t+1-\hat{S}(n_r))^2}{2\hat{S}(n_r)}}$, the number of documents is in the range (n_l, n_r).

Applications may need to ensure a certain confidence that the estimation is correct. In this case, Theorem 4.2 can also be used to compute upper and lower bounds of the set size for a given error probability.

These size estimation results do also hold for set unions created by bitwise *OR* of the respective bloom filters (cf. Section 2). However, estimation of the size of an intersection represented by a bloom filters composed with bitwise *AND* is not directly possible using theorem 4.2. The reason is that the same bits may have been set in BF_1 and BF_2 from two distinct objects, one belonging only to set S_1 and the other only to S_2. The resulting sparsity of the composite bloom filter will thus be incorrectly influenced. The probability for such a bit collision is also quite high: $(1-(1-\frac{1}{m_1})^{k_1 n_1}) * (1-(1-\frac{1}{m_2})^{k_2 n_2})$. These bits would also be set to 1 in the resulting BF_{AND}, but no element from the real intersection would set these bits in BF_{AND}. Consequently, our previously introduced estimation of elements becomes incorrect. If BF_1 and BF_2 are still available, we can estimate the size of the intersection indirectly by exploiting the fact that $|S_1 \cap S_2| = |S_1| + |S_2| - |S_1 \cup S_2|$. Thus, to estimate the intersection of S_1 and S_2, we first compute their union, and then derive the resulting estimation from the other three given bloom filters.

5 Conclusions

This work includes three techniques which improve the join efficiency in distributed query execution based on Bloom filters. These techniques are based on three new operations on bloom filters: a) creating set intersection and union (and estimating their error probabilities) from given bloom filters, b) reducing the resolution of a bloom filter while maintaining a low error probability, and c) estimating the number of elements that are hashed in a bloom filter.

Some of the scenarios for which the above operations are useful are described in this paper. More scenarios are possible, in fact, each of the proposed operations can also be integrated in existing distributed query engines for improving the performance of their query planning algorithms. Additionally, as bloom filters were already used in several application domains ranging from distributed query execution and web caching to P2P query routing and set reconciliation, we can imagine several enhancements to the existing algorithms using our extensions.

Future work: Our next task will be to align our optimizations in a fully distributed query execution engine. To do this, we currently examine several approaches on precomputing bloom filters (e.g. for all indexed keys) and/or caching them (e.g. for repeatedly occurring sub-queries), to increase join performance. Some other issues are still open, like where the bloom filters are stored and how they will be updated. A further extension is comprehensive cost-based query planning based on the estimations retrieved from the bloom filters.

References

[1] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst.*, 4(1):1–29, 1979.

[2] 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 SIGIR Conference on Research and Development in Information Retrieval*, pages 67–74, 2005.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. R. Jr. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the Tenth International Conference on Very Large Data Bases (VLDB)*, pages 323–333, 1984.

[6] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Allerton Conference*, 2002.

[7] W. chang Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1520–1529, 2001.

[8] 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)*, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[9] M.-S. Chen and P. S. Yu. Combining join and semi-join operations for distributed query processing. *IEEE Trans. Knowl. Data Eng.*, 5(3):534–542, 1993.

[10] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2003.

[11] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[12] 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.

[13] 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.

[14] A. Kumar, J. J. Xu, and E. W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *INFOCOM*, 2005.

[15] Z. Li and K. A. Ross. Perf join: An alternative to two-way semijoin and bloomjoin. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 137–144, 1995.

[16] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 829–840, 2005.

[17] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers. In *16th International Conference on Very Large Data Bases (VLDB)*, pages 198–209, 1990.

[18] 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)*, pages 149–159, 1986.

[19] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[20] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.

[21] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 110–121, 1989.

[22] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *16th International Conference on Very Large Data Bases (VLDB)*, pages 469–480, 1990.

[23] C. Wang, A. L. P. Chen, and S.-C. Shyu. A parallel execution method for minimizing distributed query response time. *IEEE Trans. Parallel Distrib. Syst.*, 3(3):325–333, 1992.