Bloom Filters in Distributed Query Execution

CSE 544 Project
Paraschos Koutris
University of Washington

1. INTRODUCTION

The MapReduce framework [5] has emerged as a successful parallel computation model in large-scale data analytics, mostly due to its simple interface and its scalability over thousands of nodes. However, while various primitives, such as aggregations, are performed efficiently in this framework, more complicated relational algebra operations such as joins and multiway joins are still implemented in a naive way. Since MapReduce provides a simple and easily programmable framework for massive parallelism, handling join computations efficiently is crucial for many applications. Some ideas in this direction have already been proposed [1] and studied. Following this trend, the purpose of this project is to consider a restricted class of optimizations for parallel computing which use the notion of bloom filters, with the goal of reducing the communication cost for multiway joins.

A join in a MapReduce environment is usually implemented with a distributed HASH-JOIN algorithm. Let us consider the natural join $R(x, y) \bowtie S(x, z)$ between relations R and S. The algorithm is equipped with a hash function, which maps values from the database domain to servers. Each of the tuples in R and S are sent to a server according to the hash value of the common x-attribute. After the mapping, tuples with the same value in the common attribute will end up in the same server; hence, each server can now locally match the tuples from the hashed partitions of R and S. Multiway joins are implemented in a naive way, that is, using consecutive MapReduce jobs.

At this point, it is natural to ask whether there are ways to improve the performance of the above algorithm in terms of the communication cost. As an example, let us consider the case of joining two relations R, S where the matched values are few and the records are relatively large. The naive HASH-JOIN algorithm sends blindly all the tuples to some server, when we could have potentially filtered a fraction of tuples through efficient preprocessing. One way to achieve this is to deploy is a standard technique in distributed query processing, which is based on *semi-join reductions* [2]: we project R on the x-attribute, send it to semi-join with S and then send only the remaining tuples to join with R. It turns out, that, instead of sending the projection of R, we can send a hashmap of R, or even better, a probabilistic structure called *bloom filter*. In this project, we will push this idea even further, exploring the situations where it is advantageous to use a bloom filter and how we can utilize it in the best way.

Structure. The structure of this project is as follows. We will first present bloom filters and define the distributed model of computation we will use, which is more general than MapReduce. Subsequently, we will discuss how bloom filters can be computed efficiently in a distributed way and then present and analyze bloom-filter based optimizations for simple and more complex join operations. Last, we will discuss various issues that occur and present some ideas for follow-up work.

Related Work. Bloom filters were first presented and used for efficient distributed join computation in [4] and [6], as part of thr R^* optimizer. The algorithm the authors propose is called BLOOMJOIN and its purpose was to reduce communication cost as data is transfered. BLOOMJOIN proved to be more efficient in reducing cost and improving performance than any other technique, e.g. semijoins. However, the setting is different from what we explore, since they assume that tables R and S are placed in different servers, but not partitioned. This allows a simpler and more efficient computation of the bloom filter. In this project, we extend the algorithm to work in a massively parallel environment, where the relations are shredded into fragments located in different servers.

Furthermore, there has been work on how to deploy bloom filters when computing more complicated multiway joins [7, 10]. In this case, the authors also assume no partitioning of the relations into fragments placed in different servers.

2. PRELIMINARIES

In this section, we first introduce bloom filters and describe their parameters in detail. Next, we present the model under which distributed computations will be studied.

2.1 Bloom Filters

Bloom filters, introduced in [3], are probabilistic data structures used for testing membership in a set. A bloom filter, in its simplest form, is essentially a bit array, equipped with k hash functions. Each hash function maps a value to some bit of the filter. More specifically, a bit is set to one *iff* some value hashes in that position, else it is zero. In order to check membership of an element, we look at the k positions where the value is hashed and answer positively only if all k bits are set to 1. This strategy allows false positives, but never false negatives.

Given a set S, we will denote by $BF(S)$ the bloom filter of S. Assuming that we want to achieve a probability of false positives bounded by p, what is the minimum size of $BF(S)$?

The exact formula for the optimal value is difficult to handle; for our purposes, we will use the approximation that we need $m_p = -\ln p/(\ln 2)^2$ bits per element of S, that is $-|S| \cdot \ln p/(\ln 2)^2$ bits in total. In practice, the approximation is very precise, as it captures the asymptotic cost of the bloom filter.

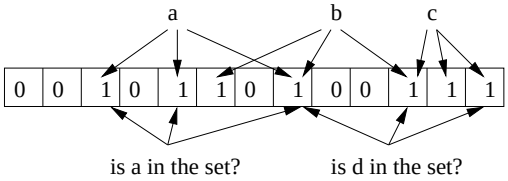

Figure 1: A bloom filter for the set $S = \{a, b, c\}$ and 3 hash functions. Asking for element d is a case of a false positive.

The advantages of using a bloom filter can be summarized as follows.

- *Space efficiency*: the size of the bloom filter is linear to the size of the set and does not depend on the universe from where S takes its values.
- *Fast construction*: constructing a bloom filter is very fast, since it requires a single scan of the data.
- *Efficient membership testing*: checking the membership of an element in S requires only computing k hash functions (where k is usually a small constant) and accessing k bits.

Naturally, there exists a tradeoff for all these advantages: the answer we get is probabilistic. However, we can regulate the probability of failure p in a flexible way. Many variants of bloom filters that have been proposed [8, 9] can achieve even better space utilization and stricter probabilistic guarantees. However, for the purposes of this project, it is sufficient to examine the applicability of bloom filters in their simplest form.

2.2 The Model

In our model, the data resides in several servers and is partitioned horizontally. Every relation R will be partitioned to fragments $R_1, \ldots, R_P$ of equal size, where P is the number of processing units available. In general, we assume that the partition is performed in an arbitrary way, but we will also study how a regular partition may help improving the algorithms we propose.

We also assume that the servers can communicate by sending tuples and information to any other server. We define a *communication step* as any number of communications which can be performed in parallel, without any synchronization barrier.

Throughout this project, we will make the standard assumption that the tuples are uniformly distributed within a relation, i.e. there is no *skew*. The main cost measure will be the communication cost, which consists of the total number of bits exchanged by the servers during the execution of the algorithm. We will also refer to the number of communication steps that an algorithm uses.

Let us also introduce some more useful parameters. For any relation $R(x, \ldots)$, let $r = |R|$. We also define by $v_x(R)$ the number of *distinct* values of the x-attribute at relation R. Moreover, let b_R be the bits of the records in R and b_x the number of bits for the key x. Finally, given a join $R \bowtie S$, we need to express the selectivity of S on R, which is the number of tuples of R that actually participate in the join. We denote this parameter by α_{SR}. We will often use $a_{SR} = 0.1$ as a typical value.

3. COMPUTING BLOOM FILTERS OF DISTRIBUTED RELATIONS

Before tackling the problem of utilizing bloom filters for distributed computations, we will deal with a somewhat independent problem. Suppose that we want to compute the bloom filter of a relation $R(x, \ldots)$ on the attribute x, where the relation is horizontally partitioned across P servers, as in our setting. To make things simpler, we will initially assume that each x-value appears only once in the relation. The goal is to store the bloom filter at a distant server with minimum communication, while achieving an error probability at most p.

We first propose and analyze two simple algorithms for this task, which we then subsume to a more generic algorithm.

Algorithm BloomOr. Each partition R_i computes a bloom filter $BF(R_i)$ for its own x-values, but $|BF(R_i)| = m_p \cdot r$, which corresponds to the size of a bloom filter for the whole relation R. After collecting the various bloom filters in the target server, we union them by a bit-wise OR operation, i.e. $BF(R) = \bigvee_i BF(R_i)$. Clearly, the union of the bloom filters is exactly the bloom filter for relation R, thus we get a probability of error equal to p. The communication cost will be $C_{OR} = m_p \cdot P \cdot r$ bits.

Algorithm BloomConcat. In this algorithm, each partition computes a bloom filter $BF'(R_i)$ such that $|BF'(R_i)| = m_{p'} \cdot r/P$, that is, the size corresponds to the size of the partition and not to $|R|$. After collecting the bloom filters in the target server, the algorithm concatenates the P small boom filters into a single one, i.e. $BF'(R) = \bigoplus_i BF'(R_i)$. In order to check the existence of a value in this filter, we must check every one of the P bloom filters; we answer negatively only if all filters answer so. This implies that we must carefully decrease the value of p' such that the final probability of error is again equal to p.

More precisely, we have to set the probability of failure p' for an individual server such that $p = 1-(1-p')^P$, or equivalently $p' = 1-(1-p)^{1/P}$. Thus, the total communication for algorithm B is $C_B = m_{p'} \cdot P \cdot r/P = m_{p'} \cdot r$.

Comparing the two algorithms, we can observe that algorithm *BloomConcat* is superior in terms of communication demands, since $m_{p'} < m_p \cdot P$. However, there is a trade-off, since using algorithm *BloomConcat* increases the look-up cost when checking for membership in the bloom filter. More precisely, the look-up time for algorithm *BloomConcat* increases linearly to P, while it remains constant for *BloomOr*. We will examine these algorithms in more detail by moving to a more general setting, which captures both algorithms as special cases.

Algorithm BloomHyb. This algorithm is essentially a hybrid of *BloomOr* and *BloomConcat* and takes as input a

parameter $k = 1, \ldots, P$, which defines a partition of the P servers into k equal-size groups (each group has P/k servers). Each server computes a bloom filter $BF^k(R_i)$ of its own partition, with size $|BF^k(R_i)| = (m_{p_k} \cdot r)/k$. The bloom filters are then transmitted and collected by the target server. For each group g, we compute the bit-wise or: $BF_g = \bigvee_{i \in g} BF^k(R_i)$. The final bloom filter is the concatenation of all the "group" filters, that is, $BF(R) = \bigoplus_g BF_g$.

It is easy to see that algorithm *BloomHyb* captures algorithm *BloomOr* when $k = 1$, and algorithm *BloomConcat* when $k = P$. It now remains to compute the communication cost, in order to guarantee a probability p of false positives. The total cost will be $C_k = m_{p_k} \cdot r \cdot (P/k)$. Moreover, we must ensure that $p = 1-(1-p_k)^k$. Consequently, we obtain the following formula:

$$C_k = -\frac{\ln(1-(1-p)^{1/k}) \cdot r \cdot P}{k \cdot (\ln 2)^2}$$

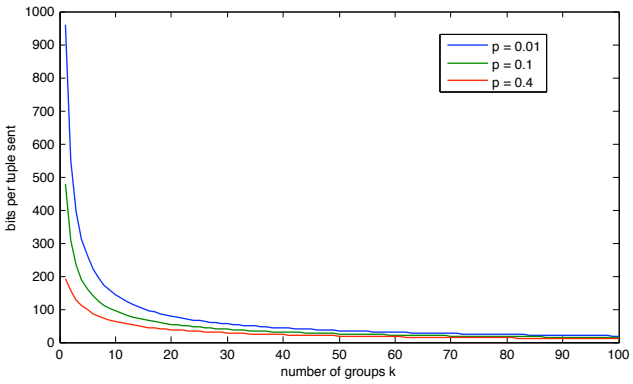

Figure 2: A plot of C_k/r as a function of k for 100 servers and 3 different values of p: 0.01, 0.1, 0.4

In figure 2, we observe a significant improvement in communication cost for relatively small values of k, for which values the look-up cost does not grow much either. The improvement in communication by increasing k even more would be negligible compared to the deterioration we would obtain in terms of access time.

In figure 3, we fix k and P and observe the relation of C_k/r (which corresponds to bits communicated per tuple) to p. Notice that getting very good probabilities gets substantially more expensive as we move closer to 0.

Can we give a rule of thumb for a reasonable value of parameter k? Since the tradeoff is between communication cost and processing time, the best value for k will depend on the parameters of the system. However, we notice that $1-(1-p)^{1/k} \approx p/k$, when p is very small. Then we obtain that the cost changes as $(\ln(k) - \ln(p))/k$. This implies that increasing from $k = 1$ to $k = 10$, we will get a decrease in communication cost in the order of 10% of the initial cost (we ignore the logarithmic factor). Moving further down to 5% of the initial cost would mean that $k = 20$, hence doubling the access cost while sightly reducing communication. To sum up, a value of k to the range of 10 to 20 would be reasonable for most cases.

We have so far considered only the case where each key-value appears only once. We will next study how to adapt our results in the case that an x-attribute assumes a specific

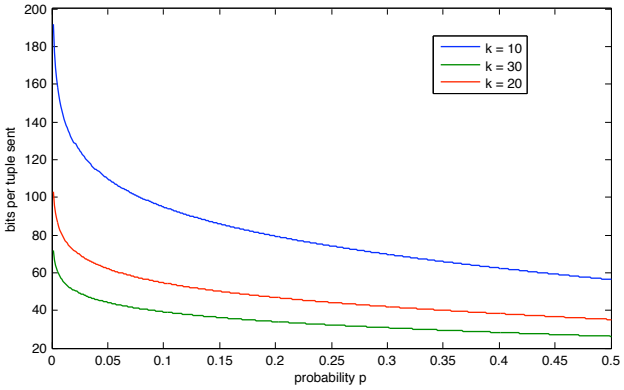

Figure 3: A plot of C_k/r as a function of p for 100 servers and $k = 10, 20, 30$

value multiple times. More specifically, since we assume a relation with uniformly distributed values over the domain, we will assume that the multiplicity of each x-value is bounded by some parameter d. Let us first examine how this changes the bloom filter construction for the easy case of algorithm *BloomOr*. In this case, we can restrict the size of $BF_x(R)$ to r/d. This means that the communication cost drops to $C'_{OR} = m_p \cdot P \cdot r/d$.

How does this affect the generic algorithm *BloomHyb*? Notice that when we have $k > 1$ groups, it may be the case that each group contains only one tuple from each value (and all the other are in the other group). Without knowing more information about the distribution of the data, we have to be conservative and thus we can not reduce the size of the bloom filters as in algorithm *BloomOr*. Hence, the performance for $k = 1$ improves as d grows, while the performance for strategies with $k > 1$ stays the same.

However, if we add the extra assumption that the relation is uniformly distributed among the servers, we can claim that each of the k groups holds d/k tuples from a particular x-value. This means that we can reduce the size of the bloom filters. Then, we have that $C'_k = C_k/\min\{1, d/k\}$. Although the increase in performance deteriorates as k increases (and is the same when $k > d$), we are still able to exploit the fact that x-values appear multiple times even for strategies with $k > 1$.

4. JOIN COMPUTATION

In this section, we will study the use of bloom filters for computing the natural join $R(x, y) \bowtie S(x, z)$. Before we present the algorithm using bloom filters, we will present the standard algorithm (HASH-JOIN) and compute its cost.

Hash-Join. The HASH-JOIN algorithm sends each tuple of R, S to a specific server, according to the hash value of the x-attribute. Then, the tuples are probed for matching locally. The total communication cost is $C = r \cdot b_R + s \cdot b_S$. The communication cost for R is $C(R) = r \cdot b_R$.

Bloom-Join. The concept of using bloom filters to minimize communication is based on the semi-join technique. The idea is that, instead of transmitting all of R, we filter a percentage of R-tuples using a lightweight data structure

which conveys useful information about S. In many cases, a large fraction of tuples from R will be rejected without being hashed. We use bloom filters as this data structure (whereas semi-joins use the projection of R to the key x).

More specifically, in order to filter R we perform the following steps: we send the bloom filter $BF_x(S)$ to R, then we filter the tuples of R with $BF_x(S)$ and last we hash the remaining tuples. Notice that we can execute the same strategy symmetrically for S. A strong advantage is that we can reason on the efficiency of filtering R or S, both of them or none of them, by examining independently the savings on the communication cost for each relation.

We have not yet described how the first step works, that is, how the algorithm computes and sends $BF_x(S)$ to every server. This is not a trivial task, since the relation is partitioned. For this, we can explore several different strategies.

Strategy A. We compute $BF_x(S)$ at some server s, by using algorithm *BloomOr*. We then broadcast the resulting bloom filter to every server. Although this strategy is communication-efficient, the computation of the bloom filter at a central server forms a bottleneck for our algorithm.

Strategy B. In order to overcome the bottleneck of the first strategy, we have to compute $BF_x(S)$ using some strategy in parallel for all P servers (e.g. algorithm *BloomOr*). This increases the communication cost by a factor of P. We can potentially mitigate this increase by using the hybrid algorithm presented in the previous section.

Cost Analysis. We now analyze strategy A. Each server s_i computes a bloom filter with $m_p \cdot v_x(S)$ bits, which is then transmitted to P servers. Thus, the cost from this step is $m_p \cdot v_x(S) \cdot P$. Next, we need to count the number of bits R is going to send after performing the filtering. R will send $\alpha_{SR} \cdot r$ tuples, since there will be α_{SR} tuples which participate in the join. Moreover, we have to account for the false positives, which correspond to $(1 - a_{SR}) \cdot p \cdot r$ more tuples for a fixed probability of error p. Thus, the total cost charged to communicating relation R is

$$C_{BF}(R) = m_p \cdot v_x(S) \cdot P + [\alpha_{SR} + (1 - a_{SR}) \cdot p] \cdot r \cdot b_R$$

which we need to compare with $C(R) = r \cdot b_R$. We will use as a measure of the bloom filter efficiency the ratio

$$g(p) = \frac{C_{BF}(R)}{C(R)} = \alpha_{SR} + (1 - a_{SR}) \cdot p + \frac{m_p \cdot v_x(S) \cdot P}{r \cdot b_R} \quad (1)$$

For ease of notation, define $c = \frac{v_x(S) \cdot P}{(1-a_{SR}) \cdot r \cdot b_R \cdot \ln^2(2)}$. Since we can choose the value of p as we want, we pick the one which minimizes the ratio and define the optimal ratio $g = \min_p g(p)$. We will now compute the optimal value for p. Equivalently, we ask to minimize the function $f(p) = \alpha_{SR} + (1 - a_{SR}) \cdot (p - c \cdot \ln(p))$. Taking $f'(p) = 0$, we obtain that $1 - c/p = 0$ or $p = c$. It is easy to see that the optimal value for p is $p_{opt} = \min\{c, 1\}$. Setting $p_{opt} = 1$ essentially means that we choose not to use the bloom filter. For the values $p < 1$, we now have a new expression

$$g = \alpha_{SR} + (1 - a_{SR}) \cdot (c - c \cdot \ln c) \quad (2)$$

A first observation about equation 2 is that the improvement we have can never surpass the selectivity of relation R.

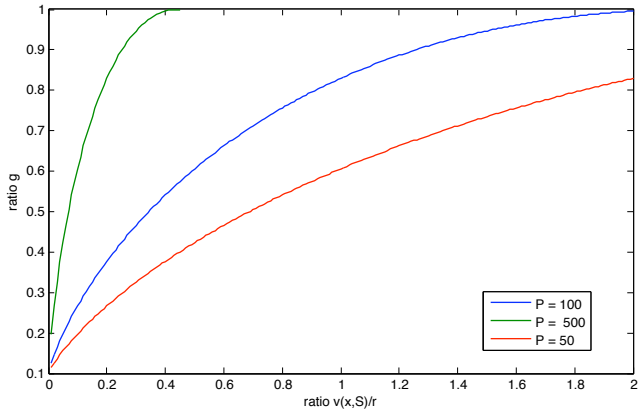

Figure 4: A plot of g as a function of the ratio $v_x(S)/r$ (distinct values of S per tuple of R) for $b_R = 512$, $a_{SR} = 0.1$ and $P = 50, 100, 500$.

By observing figure 4, one can see that when $v_x(S)$ is less than $1/5$ of $|R|$, the bloom join algorithm performs better in terms of communication cost. Specifically for the cases where the number of processors is less than 100, the bloom join outperforms the standard HASH-JOIN even when the ratio $v_x(S)/r = 2$. Moreover, one can observe that the ratio is not scale-invariant, that is, as P grows, the ratio grows to 1 very quickly. However, even with 500 nodes, when S is much smaller than R, bloom filters can still reduce communication.

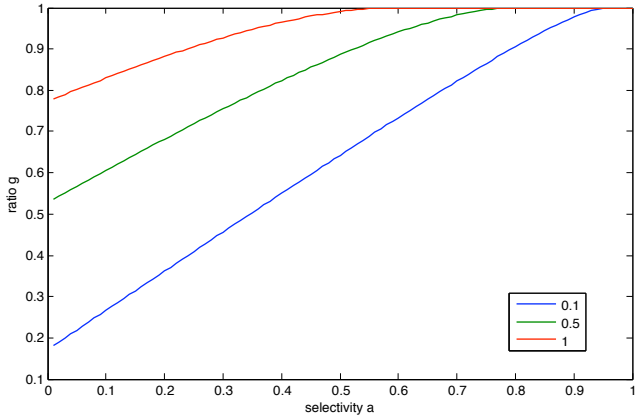

Figure 5: A plot of g as a function of the selectivity a_{SR} for $b_R = 512$, $P = 100$ and $v_x(S)/r = 0.1, 0.5, 1$.

Figure 5 shows how the ratio evolves as a function of the selectivity. First, notice that, even when the selectivity is very small, there is a considerable amount of communication, since we have to transfer the bloom filter. Second, the graphs are approximately linear to a_{SR}, which means that the algorithm can exploit the fact that the selectivity is decreased (although the slope changes according to the other parameters).

Next, we briefly turn our attention to analyzing the cost for strategy B. We use the most cost-efficient algorithm from the previous section, that is, algorithm *BloomConcat*. For this case, we can compute in a similar way that the ratio

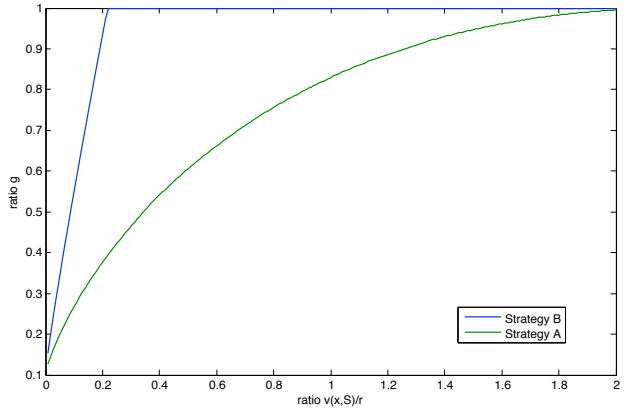

Figure 6: A plot of g as a function of the the ratio $v_x(S)/r$ for strategies A and B and parameters $a_{SR} = 0.1$, $P = 100$, $b_R = 512$.

g' will be

$$g' = \min_p \left[\alpha_{SR} + (1 - a_{SR}) \cdot p - \frac{\ln(1-(1-p)^{1/P}) \cdot v_x(S) \cdot P}{r \cdot b_R \cdot (\ln 2)^2} \right] \quad (3)$$

In figure 6, we plot g for strategies A and B as a function of the ratio $v_x(S)/r$. It is easy to observe that, even though there are points where strategy B behaves better than the standard algorithm, the ratio moves to 1 very quickly. Moreover, strategy B incurs a much higher access cost to the bloom filter.

Comparison to Semi-Join. Another question that occurs is whether a semi-join strategy would be more efficient than a distributed bloom join. In other words, what if we send $\pi_x(S)$ instead of $BF_x(S)$? For fragment i, we need to send $v_x^i(S) \cdot b_x \cdot P$ bits. Hence, the total extra communication will be $C_{SJ} = P \cdot b_x \cdot \sum_i v_x^i(S) \geq P \cdot b_x \cdot v_x(S)$. We have to compare this to $(1 - a_{SR}) \cdot (c - c \ln c)$. It is easy to see that the bloom-join strategy is almost always superior to the semi-join strategy (at least for reasonable parameters). Furthermore, filtering the tuples with $\pi_x(S)$ is much slower than using a bloom filter, since we actually need to perform a join computation.

4.1 Hashed Relations

So far, we have considered an arbitrary horizontal partition of the data residing on the servers. As we have seen, this implies that the bloom filter approach does not scale well when the number of processors increases. Here, we will investigate whether we can optimize the use of bloom filters in the case that the data is partitioned based on some locality principle.

More specifically, we consider the case where the one relation, let it be S, is partitioned according to the value of the common attribute x. In this case, the standard strategy, without the use of bloom filters, is to hash R according to the x-attribute, while S does not need to be communicated. This results in communication cost equal to $C_R = r \cdot b_R$.

Alternatively, we can try to compute $BF_x(S)$, send it to R and filter the R-tuples that we send. We will try to exploit the fact that S is regularly partitioned while computing the bloom filter. Note that each server holding S_i can compute $BF_x(S_i)$ and broadcast it. Thus, after the first communication step, each server will hold $BF_x(S_i)$ from every server i. Moreover, since the partitioning of S is known, the servers holding R know exactly which bloom filter will be used for very tuple.

The cost analysis for this case is similar to the analysis for the arbitrary partitioning. The cost of the bloom filter will be $\sum_i m_p \cdot v_x(S_i) \cdot P = m_p \cdot v_x(S) \cdot P$. Hence, the total cost will be $C_{BF} = (a_{SR} + (1 - a_{SR})p)r \cdot b_R + m_p \cdot v_x(S) \cdot P$. Though we have no cost savings, we avoid the bottleneck of computing the bloom filter in a single server before broadcasting it, which will speed up the execution of the algorithm.

5. CONJUNCTIVE QUERIES

In the previous section, we discussed how bloom filter techniques can be used for a join computation. We now consider bloom filter optimizations in the case of general conjunctive queries.

As an example, let us consider the case of a 3-way join: $R(x, y), S(y, z), T(z, w)$. Notice that S can be filtered with a bloom filter $BF_y(R), BF_z(T)$ or even both. Similarly, R can be filtered with $BF_y(S)$. However, following the concept of semi-join reducers, we can try to filter S with $BF_z(T)$, obtain S', and then compute $BF_y(S')$ and send it over to R. This strategy can potentially further increase the number of filtered tuples. Nevertheless, such a strategy adds a considerable computation overhead, for two main reasons: it requires two communication steps instead of one, which adds extra synchronization cost, and it also adds significant computation delay because S must wait to be filtered before its bloom filter is computed. Hence, considering more complex filtering strategies does not seem a viable solution. For this reason, we will consider only optimization strategies which can be performed in parallel in one communication step.

This simplifies the problem substantially. Turning our attention to the general problem, let us consider a relation R in the conjunctive query Q. This relation has common variables (i.e. joins) with some relations $S_1, \ldots, S_k$. In order to filter R, we can choose among a subset of $\{S_1, \ldots, S_k\}$, compute their bloom filters on the common variables with R and send them over to R. This can be completed in just one communication step, since we can compute and send the bloom filters in parallel.

A second important observation is that we can decide upon the bloom filter strategy we will use for R *independently* of what strategy we follow for the other relations in Q. Thus, the initial problem boils down to a much simpler question: *given a relation R and a set of joining relations $S = \{S_1, \ldots, S_k\}$, which subset of S should we use to filter R so as to minimize the communication cost?*

We will first study a simplified version of the problem, where every relation in S has similar parameters, that is, same selectivity on R and same size and size of records. Moreover, we will assume that tuples from R are *independently* filtered by each relation. In this setting, the question reduces to choosing the optimal number of relations for filtering (since they are all identical).

Let us now compute the total communication cost when we choose i relations ($i = 0, \ldots, k$) for filtering R. First, the communication cost for sending only the bloom filters will be $i \cdot m_p \cdot P \cdot v_x(S)$. In order to compute the tuples we will eventually send, we will interpret the fact that a tuple is sent as a probabilistic event which depends on the randomness

of the bloom filter and the selectivity a_{SR}. Denote by $\mathcal{B}_j$ the event that a tuple is "accepted" by the j-th bloom filter. Moreover, notice that a tuple will be sent only in the case it is accepted by all the filters. This implies that the probability that a tuple t is sent will be

$$p(t) = Pr[\wedge_j \mathcal{B}_j] = \prod_j Pr[B_j] = (a_{SR} + (1 - a_{SR}) \cdot p)^i$$

where p is the probability of error we choose for the bloom filters. Interpreting the probability as the fraction of tuples which will be sent, and denoting $c = \frac{v_x(S) \cdot P}{r \cdot b_R \cdot \ln^2(2)}$, we conclude that the ratio $g(i, p)$ for a strategy of i filters is

$$g(i, p) = (a_{SR} + (1 - a_{SR}) \cdot p)^i - i \cdot c \cdot \ln p \qquad (4)$$

Moreover, we have the constraint $0 \leq p \leq 1$. We now want to compute the optimum value of p for a given i. This can be done easily, since the derivative of $g(i, p)$ is a polynomial and its root gives the optimum value p_{opt}. For the purposes of this project, we will only plot how $g(i)$ behaves as a function of i for different values of c and different values of a_{SR}.

As we can see in figure 7, it is always better in terms of communication to use more relations to filter R. However, the gain observed is not significant compared to what we obtain using only one relation for filtering.

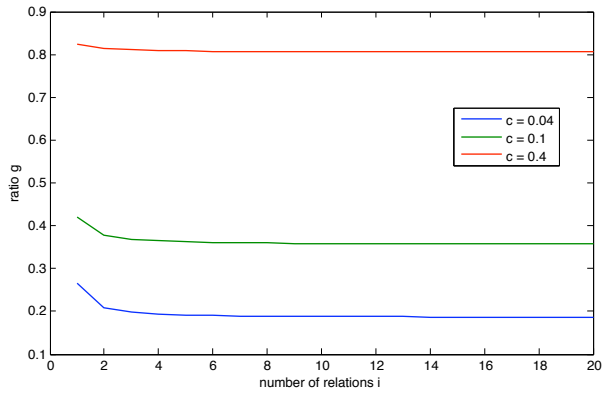

Figure 7: A plot of $g(i)$ for $a_{SR} = 0.1$ and different values of c.

In figure 8, we can observe a more advantageous use of multiple bloom filters; when the selectivity factor becomes larger, using more bloom filters drops the communication cost significantly more.

Finally, let us drop the simplifying assumption that all candidate relations for filtering are identical. This implies that we can choose to construct a bloom filter for each relation using a different error probability. Denote by p_i the error probability for relations S_i. Notice also that setting $p_i = 1$ is equivalent to not using the bloom filter of relation S_i. Using the same argument as before, we conclude that the optimal ratio g is expressed as the solution of the following optimization problem.

$$\text{minimize} \quad \prod_{i=1}^{k} (a_i + (1 - a_i) \cdot p_i) - \sum_{i=1}^{k} c_i \cdot \ln(p_i)$$

$$\text{subject to} \quad \forall i = 1, \ldots, k : 0 \leq p_i \leq 1$$

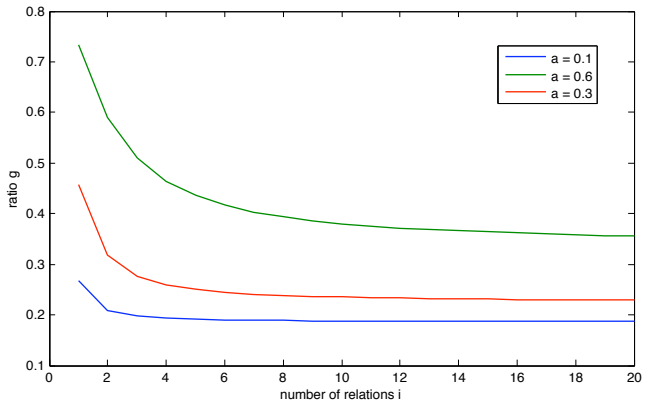

Figure 8: A plot of $g(i)$ for $c = 0.04$ and $a_{SR} = 0.1, 0.3, 0.6$.

Thus, the optimization problem is to minimize an objective function subject to some simple linear constraints. Even if we cannot afford to compute the optimal allocation of probabilities, we can potentially evaluate the function over a coarse k-dimensional grid for $p_1, \ldots, p_k$ and choose the minimum value.

6. TRANSITIVE CLOSURE

In this section, we will briefly discuss how our approach to using bloom filters may apply to a computation of the transitive closure of a relation. We consider the following datalog program, which describes the transitive closure T of a graph $G = (V, E)$

```
T(x,y) :- E(x,y).
T(x,y) :- T(x,z), E(z,y).
```

In the case that E is small enough, we could broadcast E to every server and then compute in a server s only the paths that start from a node that is hashed to server s. In the case that E is large, we have to partition G across the servers. In order to do the partitioning, we hash the tuples from E according to their first attribute. The computation is performed in an iterative way and each server is responsible for joining only the tuples in T such that their endpoint hashes on this server. This means that, as a new tuple is produced, it must be redistributed to the right server. More concretely, when a server produces a new tuple (a, b), this tuple must be sent to the server $h(b)$. There, the new tuple will be checked to see if it has already been generated by some previous iteration. If not, it will be joined with the edges that reside in the server.

Bloom filters can be useful to reducing the communication during the redistributing step. First, we compute the bloom filter $BF_y(E)$. This bloom filter essentially tries to capture all vertices that have no outgoing edges. The bloom filter is then distributed to every server. Now, when a new tuple is computed, before we redistribute it, we filter it through the bloom filter and transmit it only if it is not filtered. Intuitively, before sending the new tuple (a, b), we ask whether the vertex b has any outgoing edges. If the case is such, we do not need to send (a, b) at all.

In the case where the graph G has many nodes with no outgoing edges, using a bloom filter can potentially decrease

the communication. Moreover, since bloom filters are computed and distributed once, the extra cost is amortized over the execution time and is negligible.

7. LESSONS LEARNED

In this section, we will briefly summarize the important points in the course of studying bloom filters.

- Applying bloom filters in our distributed setting decreased communication cost for many configurations of parameters, for both simple joins and more complicated conjunctive queries (figures 4, 5, 8).
- The fact that we set the error probabilities of the bloom filters to our will proved a powerful tool, for example in section 5.
- The application of bloom filters does not scale well as the processors increase (figure 4). This implies that, as P gets larger, standard techniques may be more efficient than bloom filters. The main reason for this is that the bloom filter is broadcast to every server.
- We were unable to deploy the full reducer logic, since that would lead to a considerable overhead in preprocessing time (see section 5). Hence, we are restricted in filtering only with neighboring relations in the hypergraph, which may result to hanging tuples.
- Our approach measures only the advantage in communication cost. Creating the filters and synchronizing their transmission are also costs that we have not taken into account during this approach.

8. FUTURE WORK

The application of bloom filters in distributed query computation seems a very promising idea. As we have seen, the communication cost can be significantly reduced. However, it is not clear how the extra overhead for computing and sending bloom filters influences the processing time. It would be interesting to study the results over a practical implementation of bloom filters.

Furthermore, in section 7, we briefly discussed how bloom filters may help in computing the transitive closure of a relation. It would also be interesting to try a theoretical and experimental evaluation of this idea for this kind of computation. Finally, how could we apply bloom filters in more general recursive programs, such as datalog programs?

9. REFERENCES

[1] Afrati, F. N., and Ullman, J. D. Optimizing joins in a map-reduce environment. In *EDBT* (2010), pp. 99–110.

[2] Bernstein, P. A., and Chiu, D.-M. W. Using semi-joins to solve relational queries. *J. ACM 28*, 1 (1981), 25–40.

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

[4] Bratbergsengen, K. Hashing methods and relational algebra operations. In *VLDB* (1984), pp. 323–333.

[5] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[6] Mackert, L. F., and Lohman, G. M. R* optimizer validation and performance evaluation for distributed queries. In *VLDB* (1986), pp. 149–159.

[7] Michael, L., Nejdl, W., Papapetrou, O., and Siberski, W. Improving distributed join efficiency with extended bloom filter operations. In *AINA* (2007), pp. 187–194.

[8] Pagh, A., Pagh, R., and Rao, S. S. An optimal bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2005), SODA '05, Society for Industrial and Applied Mathematics, pp. 823–829.

[9] Putze, F., Sanders, P., and Singler, J. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics 14* (January 2010), 4:4.4–4:4.18.

[10] Ramesh, S., Papapetrou, O., and Siberski, W. Optimizing distributed joins with bloom filters. In *ICDCIT* (2008), pp. 145–156.