Optimizing Distributed Joins with Bloom Filters

Sukriti Ramesh, Odysseas Papapetrou, Wolf Siberski

Research Center L3S, Leibniz Universität Hannover
{ramesh,papapetrou,siberski}@l3s.de

Abstract. Distributed joins have gained importance in the past decade, mainly due to the increased number of available data sources on the Internet. In this work we extend Bloomjoin, the state of the art algorithm for distributed joins, so that it minimizes the network usage for the query execution based on database statistics. We present 4 extensions of the algorithm, and construct a query optimizer for selecting the best extension for each query. Our theoretical analysis and experimental evaluation shows significant network cost savings compared to the original Bloomjoin algorithm.

1 Introduction

With the advent of the Internet, the execution of database queries over the network has become commonplace. One of the main challenges in distributed query processing is efficient execution of distributed joins. This is especially important for information fusion from large-scale scientific data sources, such as Gene and medical databases. The Semantic Web which aims at Web-scale data fusion also relies on efficient distributed joins.

As network cost is the dominating cost factor in distributed join execution, prior algorithms focus primarily on its reduction. With semijoins [1], the nodes first exchange only the primary keys and attributes required for the joins, and in a second step collect the remaining attributes for answering the query. Hash-based semijoin algorithms [7] are similar to semijoins but they send compressed/hash representations of the attributes instead of complete tuples. Bloomjoins [4], a specialization of hash joins, use Bloom filters to compress the join-related attributes. This approach reduces the required bandwidth significantly and can be seen as current state of the art.

As we show in this paper, the Bloomjoin algorithm can be improved further for equi-joins by taking basic database statistics into account. Our contribution is twofold. Firstly, we present four alternative hash-based semijoin strategies and show an in-depth cost analysis for each of them. The results of this analysis are used by a query optimizer to choose the most efficient processing strategy for the join query at hand. Secondly, we show how to dynamically compute the optimal Bloom filter length for a given query based on selectivity, for each of the proposed strategies. Both contributions reduce network costs for distributed joins significantly.

Section 2 summarizes the related work, with a special focus on Bloomjoins, the basis of this work. Section 3 formalizes the problem and introduces the notation used throughout the paper. The proposed techniques, including their cost expressions, follow in sections 4 and 5. Section 6 models a query optimizer for selecting which of the proposed techniques should be used in each scenario. We finish with an experimental evaluation of the techniques and the conclusions.

2 Related work

Distributed query processing has been extensively studied in the past. A broad summary of query processing techniques for structured data at several sites is provided in [3]. This survey also shows how query processing and data replication and caching interact. Several techniques based on existing algorithms for optimizing queries in distributed databases have been proposed in [8].

Bloom Filters. The Bloom filter data structure was proposed in [2], as a space-efficient representation of sets of 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. 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 tests without the need of the original set. For any given element $e \in U$, we conclude that e is not present in the original set if at least one of the positions computed by the hash functions of the Bloom filter points to a bit still 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. Given that r elements are hashed in the filter, the probability that a membership test yields a false positive is $p \approx (1 - e^{-kr/m})^k$. The false positive probability is minimized by setting the number of hash functions to $k \approx \frac{m}{r} * \ln(2)$.

Bloomjoins. The Bloomjoin algorithm [4] was proposed in 1986 as a scheme for efficiently executing joins in distributed databases. Given tables T_1 and T_2 that reside in sites $Site_1$ and $Site_2$ respectively. For executing the equi-join $T_1 \bowtie_a T_2$ on attribute a, the Bloomjoin algorithm proceeds as follows. $Site_1$ prepares the Bloom filter BF_{T_1} of the records of T_1 by hashing $\pi_a(T_1)$ (the a values of each record), and sends it to $Site_2$. $Site_2$ uses BF_{T_1} to filter out all the records that do not belong to the Bloom filter, i.e., $T_2.a$ is not hashed in BF_{T_1}. It then sends the remaining records to $Site_1$, where the join is executed and the results computed.

The Bloomjoin algorithm is extensible to more than 2 sites, and can handle joins that include selections, e.g., $(\sigma_{z=10}T_1) \bowtie_a T_2 \bowtie_b \sigma_{y=0}T_3$. However, the algorithm does not specify how to minimize the network cost by varying the Bloom filter configuration. As we show in this work, setting a constant Bloom filter length and number of hashes is not network-efficient. The optimal configuration depends on: (a) the table structures, (b) the number of records in each table, and (c) the join selectivity.

3 Preliminaries

Problem Definition. Given a database $\mathcal{D}$ of j tables $\mathcal{D} = \{T_1, T_2, \ldots T_j\}$ distributed over j sites $\{Site_1, Site_2, \ldots Site_j\}$. We want to enable equi-join queries of any length, of the form $EQJoin = T_{i_1} \bowtie_{key_1} T_{i_2} \bowtie_{key_2} \ldots \bowtie_{key_h} T_{i_n}$, where $T_{i_1}, T_{i_2}, \ldots T_{i_n} \in \mathcal{D}$.

The proposed solutions, called *schemes*, are evaluated based on the network cost, i.e., total transfer volume for executing the distributed equi-joins. The network cost is the dominant cost factor in distributed query execution, especially for very large distributed databases like the ones found on the Internet today. In details, the schemes are required to be: (i) **optimizable for the query**, so that the query optimizer is able to configure the parameters of the scheme based on the query to minimize the overall network transfer volume, (ii) **comparable**, so that the query optimizer is able to select the optimal scheme, and (iii) **composable**, for composing several schemes to answer a query.

In this work we assume that the query initiator knows the locations and structures of the tables; resource discovery and schema matching are out of the focus of this work. The local execution of queries at each of the sites is well-handled by existing centralized DBMS, so we do not elaborate on this here.

Database Statistics. It is typical that the query optimizer estimates the network cost of each scheme by using database statistics for the participating sites. The required statistics are: (a) the number of records at each site, (b) the join selectivity of the equi-join, and, (c) the record length of each table. These statistics are among the standard statistics maintained by the DBMS, centralized and distributed. We thus assume that they are readily available to the query optimizer.

Schemes and Notations. We use schemes to represent the possible algorithms/layouts of communication in a distributed database system. The schemes are classified based on the presence or absence of a cache. We use the following notations throughout the paper:

$len(col)$	Length of column *col* in bits	$rowlen(T_x)$	Remaining row length at table T_x in bits
m	Length of the Bloom filter	k	Number of hash functions in the Bloom filter
α	Join selectivity	r	Number of records at each site

In the next section we describe two schemes without caching. The schemes that facilitate caching are described in section 5. For each scheme we present the cost analysis, which is used by the query optimizer for deciding on the query plan. For all the schemes we present the analysis for two participating sites, i.e., $N = 2$. Extending the analysis for any number of sites is straightforward. The extended equations are cumbersome, and they are not presented here.

4 Schemes Without Caching

We now present two schemes of query execution in distributed databases that do not employ a cache. The difference in the two schemes is the location of the final merging of the results for eliminating all false positives which are introduced by the Bloom filters.

Consider a scenario where the distributed database consists of N sites relevant to the query. Let them be denoted as $\mathcal{S} := Site_1, Site_2, \dots Site_N$ with tables $T_1, T_2, \dots T_N$. The result set at each site is represented as $ResultSet_1, ResultSet_2, \dots ResultSet_N$.

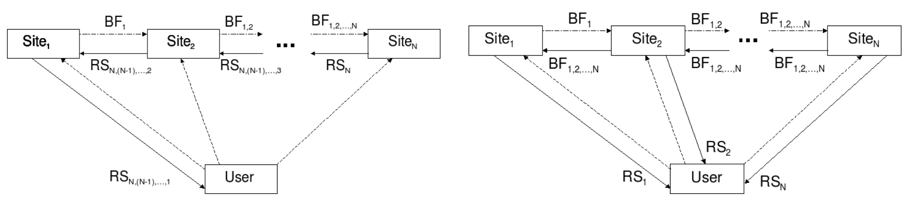

Fig. 1. (a) Scheme 1: Optimized Bloomjoin, (b) Scheme 2: Result merging at User site

4.1 Optimized Bloomjoin - Result merging at participating sites

For the execution of the query $Q : T_1 \bowtie T_2 \dots \bowtie T_N$, the optimized Bloomjoin works as follows.

Step 1. The user submits the query to the system. The query is forwarded to the N participating sites. Each of these sites, say, $Site_K$, prepares a Bloom filter BF_K relevant to the query.

Step 2. $Site_1$ sends BF_1 to $Site_2$ where $BF_{1,2}$ is computed from the bitwise-AND of BF_1 and BF_2. $BF_{1,2}$ is then sent to $Site_3$. At $Site_3$, $BF_{1,2,3}$ is created from the bitwise-AND of $BF_{1,2}$ and BF_3, and sent to $Site_4$. The same process is repeated until $BF_{1,2,3,\dots N}$ is computed.

Step 3. At $Site_N$, $BF_{1,2,3,\dots N}$ is used to retrieve $ResultSet_N$, the set of records from T_N that satisfy $BF_{1,2,3,\dots N}$. $ResultSet_N$ is sent to $Site_{N-1}$. A record join is performed between $ResultSet_N$ and the records in T_{N-1}. This gives $ResultSet_{N,N-1}$. $ResultSet_{N,N-1}$ is sent to $Site_{N-2}$. The same process is repeated until $ResultSet_{N,N-1,\dots 1}$ is retrieved.

Step 4. $ResultSet_{N,N-1,\dots 1}$ is sent to the user as query result.

Network cost. For the cost analysis, we present the case when there are two participating sites, i.e., $N = 2$. Extending the analysis for any number of sites is straightforward. The extended equations are cumbersome and they are not presented here.

The total network cost is the sum of the cost of sending BF_1 from $Site_1$ to $Site_2$, of sending $ResultSet_2$ from $Site_2$ to $Site_1$ and of sending the query results to the user.

$$\begin{aligned}Network\ Cost = Length\ of\ BF_1 + Size\ of\ ResultSet_2 * (len(key) + rowlen(T_2)) + \\ Size\ of\ ResultSet_{1,2} * (len(key) + rowlen(T_1) + rowlen(T_2)) \quad (1)\end{aligned}$$

where $len(key)$ denotes the length of the primary key and $rowlen(T_K)$ is the remaining length in bits of a record in T_K. Wlog., the number of records at both sites is r. With m we denote the length of the Bloom filter in bits. $ResultSet_2$ contains the true results of the join and the false positives supported by BF_1. Thus, minimizing the false positives is important. The probability of finding a false positive in a Bloom filter of length m is minimum when the number of hash functions is $k = m/r * ln(2)$. Then, the size of $ResultSet_2$ is:

$$\begin{aligned}Size\ of\ ResultSet_2\ &= True\ Results\ +\ False\ Positives \\ &= (Number\ of\ records\ * Join\ Selectivity) + (Record\ Set\ in\ T_2 - True\ Results) \\ &\qquad * (False\ Positive\ Probability) \\ &= (r * \alpha) + (r - r * \alpha) * (0.5)^{m*ln(2)/r}\end{aligned}$$

At $Site_1$, a record join is performed between $ResultSet_2$ and the records in T_1. This step eliminates all false positives. Accordingly, the size of the $ResultSet_{1,2}$ is estimated by:

$$Size\ of\ ResultSet_{1,2} = Number\ of\ records * Join\ Selectivity = r * \alpha$$

Then, equation(1) is rewritten as:

$$\begin{aligned}Network\ Cost = m + ((r * \alpha) + (r - r * \alpha) * (0.5)^{m*ln(2)/r}) * (len(key) + rowlen(T_2)) \\ +(r * \alpha) * (len(key) + rowlen(T_1) + rowlen(T_2)) \quad (2)\end{aligned}$$

The equation for network cost is minimized when the length of the Bloom filter gets its optimal value. We find this value by differentiating equation (2) with respect to m:

$$\begin{aligned}&\frac{d}{dm}(NetworkCost) = \\ &1 + (r * (1 - \alpha) * (0.5)^{m*ln(2)/r} * ln(0.5) * \frac{ln(2)}{r}) * (len(key) + rowlen(T_2)) \quad (3)\end{aligned}$$

To obtain m such that network cost is minimal, the left-hand side of equation(3) is set to 0. By solving the resulting equation we get the optimal value of m as,

$$m = \frac{r}{(ln(2))^2} * (ln(1 - \alpha) + 2 * ln(ln(2)) + ln(len(key) + rowlen(T_2))) \quad (4)$$

Extending the equation for more than 2 sites is straightforward. This equation is used by the query optimizer to estimate the expected cost of this scheme, and pick the best scheme for answering the query.

4.2 Result merging at User site

Given the query $Q : T_1 \bowtie T_2 \ldots \bowtie T_N$, the second scheme in which result merging is performed at the user site works as follows.

Step 1. The user submits the query to the system. The query is forwarded to the N participating sites. Each of these sites, say, $Site_K$, prepares a Bloom filter BF_K relevant to the query.

Step 2. $Site_1$ sends BF_1 to $Site_2$ where $BF_{1,2}$ is computed from the bitwise-AND of BF_1 and BF_2. $BF_{1,2}$ is then sent to $Site_3$. At $Site_3$, $BF_{1,2,3}$ is computed from the bitwise-AND of $BF_{1,2}$ and BF_3. The process is repeated until $BF_{1,2,3,...N}$ is computed.

Step 3. At $Site_N$, $BF_{1,2,3,...N}$ is used to retrieve $ResultSet_N$, the set of records from T_N that satisfy $BF_{1,2,3,...N}$. $ResultSet_N$ is sent to the user site. $BF_{1,2,3,...N}$ is sent to $Site_{N-1}$ and is used to retrieve $ResultSet_{N-1}$, the set of records from T_{N-1} that satisfy $BF_{1,2,3,...N}$. $ResultSet_{N-1}$ is sent to the user site. The same process is repeated at all N sites.

Step 4. At the user site, a record join is executed: $ResultSet_{N,N-1,...1} := ResultSet_N \bowtie ResultSet_{N-1} \ldots \bowtie ResultSet_1$, and presented to the user.

Network Cost. For the analysis we consider a distributed database setup with two participating sites, i.e., $N = 2$. The network cost for this scheme is as follows.

$$Network\ Cost = Length\ of\ BF_1\ +\ Size\ of\ ResultSet_2\ *(len(key) + rowlen(T_2)) \\ +Length\ of\ BF_{1,2}\ +\ Size\ of\ ResultSet_1\ *\ (len(key) + rowlen(T_1))$$

We denote the length of BF_1 as m_1 bits and the length of $BF_{1,2}$ as m_2 bits.

$$Network\ Cost = m_1 + ((r * \alpha) + (r - r * \alpha) * (0.5)^{m_1 * ln(2)/r}) * (len(key) + rowlen(T_2)) \\ + m_2 + ((r * \alpha) + (r - r * \alpha) * (0.5)^{m_2 * ln(2)/r_2}) * (len(key) + rowlen(T_1)) \quad (5)$$

where r_2 represents the size of $ResultSet_2$. We use differentiation to minimize Equation(5) with respect to m_1 and m_2. The values of m_1 and m_2 at which network cost is minimal.

$$m_1 = \frac{r}{(ln(2))^2} * (ln(1-\alpha) + 2 * ln(ln(2)) + ln(len(key) + rowlen(T_2))) \quad (6)$$

$$m_2 = \frac{r_2}{(ln(2))^2} * \left(ln(1-\alpha) + 2 * ln(ln(2)) + ln(len(key) + rowlen(T_1)) + ln(\frac{r}{r_2}) \right) \quad (7)$$

5 Schemes With Caching

Network *caches* are often used in distributed databases for reducing network usage. In our work we use the network cache to cache, at a single site, frequently-requested Bloom filters of tables instead of requesting them every time a join is needed. The cache is initially empty, and the coordinator decides which Bloom filters need to be cached and which should be refetched each time (see Section 6).

Fetching the Bloom filters and updating the cache. Fetching a Bloom filter uncompressed requires m bits, where m is the length of the filter. The cost is reduced by compressing the Bloom filter [5]. The expected network cost for retrieving a Bloom filter BF with compression is: $Compress(BF) = m * H(BF)$, where $H(BF)$ denotes the information entropy of the Bloom filter: $H(BF) := -Truebits/m * \log_2(Truebits/m) - (1-Truebits/m) * log_2(1-Truebits/m)$, and $Truebits$ is the number of bits set to true in BF.

Every time a cached Bloom filter is invalidated, the cache holder requests the new Bloom filter from the site which holds the table. Let the cached Bloom filter be denoted by BF_{cached} and the new one by BF_{new}. The table holder decides what is less expensive: (a) to send the Bloom filter representing the difference between the cached and the new Bloom filters: $BF_{diff}(BF_{new}, BF_{cached}) := XOR(BF_{new}, BF_{cached})$, or, (b) to send the new Bloom filter BF_{new}. In both the cases the Bloom filter is compressed before transmission. Thus, the site that holds the table selects and sends the Bloom filter with lower entropy.

The cache can reside either at one of the participating sites or at the coordinator. Although caching at a participating site is always less expensive than caching at the coordinator site, the former is not always possible; the existence and location of the cache depends on the database policies and is set by the database administrator manually. The optimizer then selects the optimal scheme based on the caching policies. We now describe and analyze both the caching approaches.

5.1 Caching at a participating site

For this scheme a site participating in the join, say, $Site_C \in \mathcal{S}$, maintains a cache of Bloom filters. For the execution of the query $Q : T_1 \bowtie T_2 \ldots \bowtie T_N$, the scheme works as follows.

Step 1. The cached Bloom filters $BF_{cached_1}, BF_{cached_2}, \ldots BF_{cached_N}$ are updated by sending $Compress(BF_{new})$ or $Compress(BF_{diff}(BF_{new}, BF_{cached}))$, whichever yields a lower network cost. The updated and cached Bloom filters are now denoted as $BF_1, BF_2, \ldots BF_N$.

Step 2. At the cache site $Site_C$, a bitwise-AND operation is performed on the updated Bloom filters $BF_1, BF_2, \ldots BF_C, \ldots BF_N$ resulting in the final Bloom filter, $BF_{1,2,3 \ldots N}$.

Step 3. $BF_{1,2,3 \ldots N}$ is compressed and sent to all the participating sites.

Step 4. At each participating site, $Site_K$, where $K = 1, 2, ...N$, $BF_{1,2,3 \ldots N}$ is used to retrieve $ResultSet_K$, the set of records that satisfy $BF_{1,2,3 \ldots N}$ at $Site_K$. $ResultSet_K$ is sent to the cache site, $Site_C$.

Step 5. At the cache site, a record join is performed: $ResultSet_{1,2,3 \ldots N} := ResultSet_1 \bowtie ResultSet_2 \ldots \bowtie ResultSet_N$.

Step 6. $ResultSet_{1,2,3 \ldots N}$ is sent to the user as query result.

Network Cost. We now present the cost analysis of this scheme for the case with two sites, $Site_1$ and $Site_2$, having tables T_1 and T_2 respectively. In our example, $Site_2$ is also the cache holder. The network cost for the scheme is:

$$\begin{aligned}
Network\ Cost &= Cost\ of\ caching\ or\ updating\ BF_1 + Length\ of\ compressed\ BF_{1,2} \\
&+ \quad Size\ of\ ResultSet_1 * (len(key) + rowlen(T_1)) \\
&+ \quad Size\ of\ ResultSet_{1,2} * (len(key) + rowlen(T_1) + rowlen(T_2))
\end{aligned}$$

where $len(key)$ is the length in bits of the primary key and $rowlen(T_K)$ is the remaining length in bits of a record in T_K. Since Bloom filters are always compressed before being sent, the cost of sending a Bloom filter BF compressed over the network is $Compress(BF) = m * H(BF)$. The expected value of the network cost is:

$$\begin{aligned}
&E(Network\ Cost) = \\
&\quad Min(Compress(BF_{new_1}), Compress(BF_{diff}(BF_{new_1}, BF_{cached_1})) + Compress(BF_{1,2}) \\
&\quad + \Big((r * \alpha) + (r - r * \alpha) * (0.5)^{m*ln(2)/r}\Big) * (len(key) + rowlen(T_1) \\
&\quad + (r * \alpha) * (len(key) + rowlen(T_1) + rowlen(T_2)) \qquad (8)
\end{aligned}$$

For the first execution of a query, the entire Bloom filter is sent from the site to the cache. Likewise, when the Bloom filters are already in the cache and are updated using BF_{new}, the entire new Bloom filter is sent from site to cache. The expected network cost corresponding to both these scenarios is expressed using equation(9).

$$\begin{aligned}
&Network\ Cost = Compress(BF_{new_1}) + Compress(BF_{1,2}) \\
&\quad + \Big((r * \alpha) + (r - r * \alpha) * (0.5)^{m*ln(2)/r}\Big) * (len(key) + rowlen(T_1)) \\
&\quad + (r * \alpha) * (len(key) + rowlen(T_1) + rowlen(T_2)) \qquad (9)
\end{aligned}$$

We use differentiation to minimize equation(9) with respect to m. The Bloom filter length that minimizes the network cost is found by equation(10).

$$\begin{aligned}
m = \frac{r}{(ln(2))^2} * (ln(1-\alpha) + 2 * ln(ln(2)) - ln(H(BF_{new_1}) + H(BF_{1,2}))) \\
+ \frac{r}{(ln(2))^2} * (ln(len(key) + rowlen(T_1))) \qquad (10)
\end{aligned}$$

Estimating the bloom filter entropy requires the number of true bits in each bloom filter. The coordinator uses the selectivity of the join to estimate the number of true bits in the Bloom filter $BF_{1,2}$, and the required entropy values, as presented in [6].

5.2 Caching at coordinator site

In this scheme the coordinator site, $Site_C$, holds the cache of Bloom filters. Given a setup of N sites $\mathcal{S}$. The coordinator site $Site_C$ does not belong in $\mathcal{S}$. The query $Q : T_1 \bowtie T_2 \ldots \bowtie T_N$ is executed as follows:

Step 1. The cached Bloom filters $BF_{cached_1}, BF_{cached_2}, \ldots BF_{cached_N}$ are updated by sending $Compress(BF_{new})$ or $Compress(BF_{diff}(BF_{new}, BF_{cached}))$, whichever yields a lower network cost. The updated and cached Bloom filters are now denoted as $BF_1, BF_2, \ldots BF_N$.

Step 2. At the coordinator site, in the cache, a bitwise-AND operation is performed on $BF_1, BF_2, \ldots BF_N$ resulting in the final Bloom filter, $BF_{1,2,3\ldots N}$.

Step 3. $BF_{1,2,3\ldots N}$ is compressed and sent to all the participating sites.

Step 4. At each participating site, $Site_K$, where $K = 0, 1, 2, \ldots N$, $BF_{1,2,3\ldots N}$ is used to retrieve $ResultSet_K$, the set of records that satisfy $BF_{1,2,3,\ldots,N}$ at $Site_K$. $ResultSet_K$ is sent to the coordinator.

Step 5. At the coordinator, a record join is performed: $ResultSet_{1,2,3\ldots N} := ResultSet_1 \bowtie ResultSet_2 \ldots \bowtie ResultSet_N$.

Step 6. $ResultSet_{1,2,3\ldots N}$ is sent to the user as query result.

Network Cost. To analyse the network cost for the scheme, we assume two participating sites, $Site_1$ and $Site_2$. The cache is present at an independent coordinator site. The network cost for the scheme with 2 sites is:

$$\begin{aligned}
&Network\ Cost = Cost\ of\ caching\ or\ updating\ BF_1 + Cost\ of\ caching\ or\ updating\ BF_2\\
&\quad + 2 * Length\ of\ compressed\ BF_{1,2} + Size\ of\ ResultSet_1 * (len(key) + rowlen(T_1))\\
&\quad + Size\ of\ ResultSet_2\ *\ (len(key) + rowlen(T_2))\\
&\quad + Size\ of\ ResultSet_{1,2} * (len(key) + rowlen(T_1) + rowlen(T_2))
\end{aligned}$$

The expected value of the network cost is:

$$\begin{aligned}
&E(Network\ Cost) = Min(Compress(BF_{new_1}), Compress(BF_{diff}(BF_{new_1}, BF_{cached_1}))\\
&\quad + Min(Compress(BF_{new_2}), Compress(BF_{diff}(BF_{new_2}, BF_{cached_2}))\\
&\quad + 2\ *\ Compress(BF_{1,2})\\
&\quad + \Big((r * \alpha) + \ (r\ -\ r * \alpha) *\ (0.5)^{m*ln(2)/r}\Big)\ *\ (len(key) + rowlen(T_1))\\
&\quad + \Big((r * \alpha) + \ (r\ -\ r * \alpha) *\ (0.5)^{m*ln(2)/r}\Big)\ *\ (len(key) + rowlen(T_2))\\
&\quad + (r * \alpha) *\ (len(key) + rowlen(T_1) + rowlen(T_2))
\end{aligned} \tag{11}$$

When tables are requested for the first time, entire Bloom filters need to be sent from the sites to the cache. Also, when the cached Bloom filters are updated using the new Bloom filters, BF_{new_1} and BF_{new_2} need to be sent from the sites to the cache. The network cost for both the above scenarios is,

$$\begin{aligned}
&E(Network\ Cost) = Compress(BF_{new_1})\ +\ Compress(BF_{new_2})\ +\ 2 * Compress(BF_{1,2})\\
&\quad + \Big((r * \alpha) + \ (r\ -\ r * \alpha) *\ (0.5)^{m*ln(2)/r}\Big)\ *\ (len(key) + rowlen(T_1))\\
&\quad + \Big((r * \alpha) + \ (r\ -\ r * \alpha) *\ (0.5)^{m*ln(2)/r}\Big)\ *\ (len(key) + rowlen(T_2))\\
&\quad + (r * \alpha) *\ (len(key) + rowlen(T_1) + rowlen(T_2))
\end{aligned} \tag{12}$$

Differentiating, we find the correspoding optimal Bloom filter length which minimizes the network cost:

$$m = \frac{r}{(ln(2))^2} * (ln(1-\alpha) + 2 * ln(ln(2)) - ln(H(BF_{new_1}) + H(BF_{new_2}) + 2 * H(BF_{1,2}))) + \frac{r}{(ln(2))^2} * (ln(2 * len(key) + rowlen(T_1) + rowlen(T_2))) \quad (13)$$

6 Scheme Comparison - A Query Optimizer for Distributed Databases

It is standard in distributed databases to assign the responsibility of query planning to a node, or a small number of participating nodes, the coordinators. The coordinators in our proposal are responsible for: (a) receiving and parsing the queries, (b) rewriting, (c) optimizing, and (d) sending the queries to the participating sites and coordinating the query execution.

The first two steps in our schemes do not differentiate from existing distributed query algorithms. The crucial step is optimizing the queries to reduce the network cost. The query optimizer breaks the query to a series of equi-joins which can be efficiently handled, and other joins, e.g., inequality joins. Then, for each equi-join it decides whether a caching scheme is more beneficial than a scheme without caching. This is decided based on the popularity of the requests for each Bloom filter, on the maximum caching size, and, in case the Bloom filter is already cached, on the change rate of the Bloom filter between requests (percentage of changed bits between each Bloom filter request).

After the optimizer decides whether each Bloom filter should be cached or not, it enumerates all the possible plans, and computes the expected cost for each of them according to the cost equations for the optimal length (equations 2, 5, 9, 11). Finally, it selects the best order for executing the query, forwards the query plan to the participating nodes, and coordinates the query execution.

7 Experimental Evaluation

The purpose of the experimental evaluation was twofold: (a) to experimentally verify the theoretical costs for the proposed schemes, and, (b) to test the importance of the optimizer in different setups. The experiments verified the theoretical cost estimations. Next paragraphs present the details on the experiments on the importance of the optimizer.

The experiments were performed on a vertically fragmented database. We created 2 tables of the following structure: `Table Personnel: int personid, char[16] name`, at $Site_1$, and `Table Professors: int personid, char[16] department`, at $Site_2$. Both the tables had a primary key length of 32 bits and remaining row length of 128 bits, and $Professors.personid$ was a foreign key of $Personnel.personid$. We then filled the two tables with 1000 records each, at each experiment varying the join selectivity from 0 to 1.

In the first experiment we compared the optimized Bloomjoin with the original Bloomjoin algorithm [4]. The original Bloomjoin was executed with Bloom filter lengths of 1000 bits, 30000 bits, and 50000 bits. The hash functions in each case were set to minimize the false positive probability: $k = m/r * \ln(2)$, where m was the length of the filter and r the number of records. In all cases the Bloom filters were compressed before sending. Figure 2(a) plots the network resources required by each of the approaches for varying join selectivity. The optimized Bloomjoin scheme is significantly better than all the constant-length Bloom filter solutions. The 1000 bits Bloom filter has an increased error probability and gives too many false positives. The effect of false positives is more visible when the selectivity is low. When the selectivity is high, the number of false positives is reduced independent of the Bloom filter length, since most of the sent records already belong to the results. The larger Bloom filters are also suboptimal since they add an unnecessary cost to the query execution.

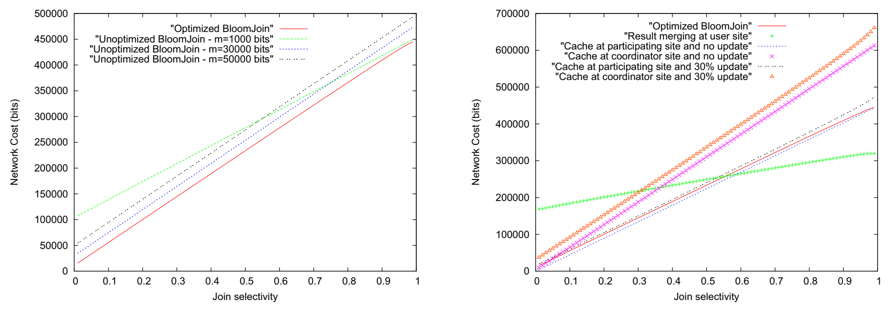

Fig. 2. (a) Optimized Vs Original Bloomjoin, (b) Comparison of all the schemes

In the second experiment we compared all the proposed schemes. Since caching depends on the update rate of the records in the tables, the schemes with caching were repeated twice, once without updating the records, and another time with an update rate of 30%. The updating was simulated in the following manner: (a) the cache holder cached all the Bloom filters, (b) the table holders were replacing 30% of their records with an equal number of new records and regenerating their Bloom filters, and (c) the cache holder was updating the Bloom filters in the cache and executing the query. Figure 2(b) plots the network cost for varying join selectivity. For low selectivity and no updates, caching at a participating site, caching at the coordinator, and the optimized Bloomjoin schemes are almost equally efficient. For high selectivity, the result set which needs to be transmitted twice makes the schemes with caching at coordinator less beneficial. Among the scenarios with updates, the optimized Bloomjoin is optimal for $\alpha \leq 0.5$. For larger join selectivities, the scheme with result merging at user site is significantly better than the others.

8 Conclusions and Future Work

Efficient algorithms for distributed joins are required for a wide range of Internet-based applications, like peer-to-peer systems and web-based distributed databases. In this work we proposed and theoretically analyzed four distributed join schemes which make use of Bloom filters to reduce network costs significantly. An integral part of our contribution is the query optimizer, which picks the optimal scheme for each query and configures it for minimizing the network usage. The optimization process involves only statistics that are maintained by default in all DBMS systems.

In addtion to the theoretical analysis, we experimentally evaluated the proposed schemes and compared them with previous work. The experimental results validate our analystical findings and show the importance of selecting the right scheme and configuring it with the right parameters. Significant reduction of the cost, more than 50%, was observed in some setups just by optimizing Bloom filter length and number of hash functions.

Our current focus is on further enhancing the analysis with network statistics, so that fast links are preferred over slower links. Sending a Bloom filter from USA to Hannover, Germany is more expensive than communicating the same filter from Munich to Hannover. Our current analysis does not yet take this into account. By including the network distance (in terms of bandwidth and/or latency) in the analysis, we will be able to reorder the joins so that usage of expensive links is minimized.

References

1. Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.
2. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
3. Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
4. Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 84–95. ACM Press, 1986.
5. Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
6. Odysseas Papapetrou, Loizos Michael, Wolfgang Nejdl, and Wolf Siberski. Additional analysis on bloom filters. Technical report, Division of Engineering and Applied Sciences, Harvard University, L3S Research Center, Leibniz Universität Hannover, 2007.
7. Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.*, 9(1):133–161, 1984.
8. Clement T. Yu and C. C. Chang. Distributed query processing. *ACM Comput. Surv.*, 16(4):399–433, 1984.