PERF Join: An Alternative To Two-way Semijoin And Bloomjoin

Zhe Li Kenneth A. Ross*

Department of Computer Science, Columbia University, New York, NY 10027

{li,kar}@cs.columbia.edu

Abstract

This paper presents "Positionally Encoded Record Filters" (PERFs) and describes their use in a distributed query processing technique called PERF join. A PERF is a novel two-way join reduction implementation primitive. While having the same storage and transmission efficiency as a hash filter (e.g., Bloom Filter), a PERF is based on the relation tuple scan order instead of hashing. Hence it doesn't suffer any loss of join information incurred by hash collisions. Using the query response time measured in terms of network cost as a comparison criterion, we demonstrate through analytical studies that PERF join performs significantly better than two-way Bloomjoin and two-way semijoin variants under a wide range of relevant cost parameter values. For the large number of distributed query processing algorithms relying on Bloomjoin or semijoin variants to reduce their network cost, we can sometimes gain an instant improvement in their response time by switching to PERF join instead. Other salient features of PERF join include cheap local join processing cost, and the capability to handle inequality and cyclic join queries.

1 Introduction

With present day technology, one major cost factor of distributed join query processing is the overhead of network communication, particularly over slow wide-area links. Thus, distributed query processing strategies attempt to minimize the amount of data transmitted over the network. A popular technique for reducing network transmission volume is the use of *semijoins*.

*This research was supported by NSF grants IRI-9209029 and CDA-90-24735, by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, and by an NSF Young Investigator award.

Semijoins were introduced in [BC81, BG81]. Suppose two relations R and S are stored at different sites. A semijoin from relation R to relation S (written $S \ltimes R$) is implemented as follows: (a) Project R on the join attributes of R and S to get a new relation P_R, then (b) ship P_R to the site of S and (c) perform the join of P_R with S. The result S' of this join is often much smaller than the original relation S, while it still contains all the information necessary to construct the full join with R. S' can then be sent to R's site (or to a third site) to construct the full join.

Most distributed query processing algorithms proposed so far [HY79, AHY83, YC84, RK91, WCS92, CY93] rely on some variants of the semijoin technique to reduce their network cost. They typically process a distributed query in the following manner:

1. Initial local processing: All local selection, projection and local join operations are performed first.

2. Semijoin reduction: After the preprocessing by the first step, the only operations left are distributed joins among the remote relations. A cost-effective semijoin program is then derived and executed (usually in sequential steps, but possibly in parallel) to reduce the size of the joining relations involved. Often both the local join processing cost and network transmission cost can be substantially reduced.

3. Final assembly: All the reduced relations that are needed to compute the query result are shipped (possibly in parallel) to an assembling site (usually the query site) where the join result is composed.

An important extension of the semijoin operation is the "two-way" semijoin, which we abbreviate as 2SJ [Dan82]. In the terminology of the description above, an additional step is performed: S' is projected onto the join attributes to get a new relation $P_{S'}$. $P_{S'}$ is sent back to the site of R. R can be joined with this projection to reduce its size and thus reduce the cost of later transmitting R. Two-way semijoin aims to reduce both R and S so that they each contain only tuples that match the other relation, while spending less total network cost than executing $R \ltimes S$ and $S \ltimes R$ in parallel. It is particularly useful when both R and S need to be transmitted to a third site, either because that is where the answer is needed, or because a third joining relation is stored at a third site. The 2SJ technique has been enhanced in [RK91] to send back to R's site the smaller of $P_{S'}$ and $P_R - P_{S'}$. We call this improved version 2SJ+.

In contrast to semijoin variants which always transmit join attribute values, Bloomjoin (hashed semijoin) [ML86] transmits Bloom filters [Blo70, Bab79] which are more space efficient. A Bloom filter is a hashing based bit vector used to encode the same joining information as the join attribute projections do. When joining R with S, the join attribute values of R are hashed to some addresses in the Bloom filter whose corresponding bits are then set to 1. A zero bit after hashing would indicate that no join attribute value that hashes to that bit participates in the join. Compared with semijoin, Bloomjoin can sometimes have lower network overhead and local join cost at the receiving site because a Bloom filter is generally smaller than the join attribute projection. Due to the nature of hash collisions present in a Bloom filter, Bloomjoin can be viewed as a lossy implementation of semijoin. That is, a relation being reduced by the Bloom filter of another relation can still contain unmatched tuples which would otherwise be eliminated by a semijoin.

This paper presents "Positionally Encoded Record Filters" (PERFs) and describes their use in a distributed query processing technique called PERF join. A PERF is a novel two-way join reduction implementation primitive. The basic idea of PERF join is as follows: As in 2SJ, S is reduced by a semijoin with P_R. But instead of transmitting $P_{S'}$ back to R, send a bit-vector (PERF) that contains one bit for every tuple in P_R. That bit is set to 1 if it is in $P_{S'}$ and 0 otherwise. The order of the bits in this bit vector is the same tuple order of P_R that R's site sent initially. Assuming that R's site can easily reconstruct this order, the value of $P_{S'}$ can be constructed at R's site from the bit vector and P_R. We will explain in section 5 how to reconstruct this order.

PERF is designed to minimize the cost of the "backward" phase of 2SJ. In many cases the PERF will be significantly smaller that the original $P_{S'}$. For example, suppose the cardinality of P_R is 100000, that the cardinality of $P_{S'}$ is 50000, and that the size of the join attribute in these relations is 4 bytes. Then the size of $P_{S'}$ is 200000 bytes, while the size of the PERF is only 12500 bytes. It is easy to see that the PERF is a lot smaller than $P_{S'}$, and negligible in size compared to the cost of sending P_R in the first place (400000 bytes). In other words, the cost of the "backward" reduction is much smaller than the cost of the "forward" reduction.

While having similar storage and transmission efficiency as a hash filter (e.g., Bloom Filter), PERF is based on the relation tuple scan order instead of hashing, hence doesn't suffer any loss of join information incurred by hash collisions. Using the query response time measured in terms of network cost as a comparison criterion, we demonstrate through analytical studies, that PERF join performs significantly better than two-way Bloomjoin and two-way semijoin variants under a wide range of relevant cost parameter values. For the large number of distributed query processing algorithms relying on Bloomjoin or semijoin variants (either one-way or two-way) to reduce their network cost, we can sometimes gain an instant improvement in their response time by switching to PERF join instead. Other salient features of PERF join include cheap local join processing cost, and the capabilities to handle inequality and cyclic join queries.

The rest of the paper is organized as follows. Section 2 lays down the terminology and assumptions adopted throughout this paper. In Section 3 we formally introduce PERFs and the PERF join operation. We then describe several semijoin variants and Bloomjoin, analyze their pros and cons, and show how PERF join can be used to either enhance or combine with them to derive more efficient query plans (e.g, with lower network and local join processing cost). A detailed analysis is given to demonstrate that PERF join outperforms its competitors. In Section 4 we quantitatively compare the performance improvements that can be achieved by adopting PERF join over other competitors through a few representative examples. We briefly discuss the maintenance issues related to PERF join in Section 5, and conclude our work in Section 6.

2 Terminology and Assumptions

When talking about the join between two relations, we shall refer to them as R and S. If applicable, we shall refer to $S \ltimes R$ and $R \ltimes S$ as the "forward" and the "backward" semijoin reduction phases, respectively.

When talking about an n-way join query, we shall name the relations as $R_1, \ldots, R_n$, and assume they are located at n distinct sites $1, \cdots, n$.

Unless otherwise stated, we shall assume the three-step paradigm described in Section 1 is followed to process distributed queries, i.e., every R_i would be reduced through some join reduction operations originated from a subset of the remaining joining relations, and the reduced R_i will be shipped to the final assembling site to compute the query result. However, no assumption is made about the homogeneity and autonomy of these sites.

For simplicity of analysis and presentation, we shall ignore the local processing cost and use the query response time measured solely in terms of the network cost as the comparison criterion for all the analysis. This is reasonable for a low bandwidth wide-area network setting. Furthermore, we shall calculate the network cost as the number of bits transmitted.

We list below some notations we use throughout this paper:

$\lvert R_i \rvert$	cardinality of R_i
$W(R_i)$	width of a tuple in R_i (in bits)
X_i	join attribute(s) of R_i
$W(X_i)$	width of the join attribute(s) X_i (in bits)
R^X	projection of R onto attribute(s) X, eliminating duplicates
$\rho(R_i)$	semijoin selectivity of $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ over R_i

Logarithms are all base 2.

For $R \bowtie S$, we use $\rho(R)$ and $\rho(S)$ to denote the *semijoin selectivity* of $R \bowtie S$ over R and S, respectively. We have $\rho(R) = |R \ltimes S|/|R|$, and $\rho(S) = |S \ltimes R|/|S|$. In general, $\rho(R_i)$ is defined as:

$$\rho(R_i) \quad = \quad \frac{|R_i \ltimes (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)|}{|R_i|}$$

3 PERF join

In this section we first introduce PERFs (an acronym for "Positionally Encoded Record Filters") and PERF join, as proposed in [LR94]. We then describe several popular semijoin variants and Bloomjoin, and show how PERF join outperforms them in terms of both network and local join processing cost.

Definition 3.1: Consider two relations R and S, with R physically ordered in some fashion. S contains the join reduction information for R. The PERF of relation R with respect to S, denoted as PERF(R), is a bit vector of $|R|$ bits. The jth bit of PERF(R) is set if and only if the jth tuple of R appears in the join result[1] $R \bowtie S$. □

As an example, Figure 1 shows two PERFs for relations $R(A,X)$ and $S(X,B)$. For instance, the 5th bit set in PERF(R) indicates that the 5th R tuple will participate in $R(A,X) \bowtie S(X,B)$. PERF(R) and PERF(S) encode the complete join reduction information of $R \bowtie S$.

R(A,X)

	A	X
1	a1	10
2	a2	20
3	a3	30
4	a4	40
5	a5	50
6	a6	60

S(X,B)

X	B	
100	b1	1
30	b2	2
50	b3	3
90	b4	4
20	b5	5
70	b6	6

PERF(R)

1	0
2	1
3	1
4	0
5	1
6	0

PERF(S)

0	1
1	2
1	3
0	4
1	5
0	6

Figure 1: PERFs for R and S

Definition 3.2: Let R and S be relations at distinct sites in a distributed database. The PERF join R with S consists of (a) sending R^X to the site of S, (b) performing $S \ltimes R$, and (c) sending PERF(R) back to the site of R to reduce R. □

If there is a third query site involved, PERF join also applies. R^X and S^X can be sent in parallel to the query site, where R^X and S^X would be joined, PERF(R) and PERF(S) (with respect to $R^X \bowtie S^X$) would then be composed and sent back to their corresponding sites to reduce both R and S.

Using the example of Figure 1, R^X and S^X would be shipped to an assembling site and joined. Once a matching join attribute value is found, its position within the corresponding join attribute projection will be used to set the bit in its PERF. For instance, the join attribute value 20 is a match, so the 2nd bit and the 5th bit will be set to 1 in PERF(R) and PERF(S) respectively.

Bloom filters on a relation R perform best when they are of size $k|R|$ where k is a small constant greater than 1 [Mul90]. From Definition 3.1, one can observe that a PERF has at least the same storage and transmission efficiency as a hash filter (e.g., Bloom filter). On the other hand, PERF is based on the tuple scan order instead of hashing. Hence it doesn't suffer any loss of join information incurred by hash collisions. These are the most obvious advantages of PERF join over other semijoin variants.

Another observation we can make is that PERFs are generated after applying a real join among the join attribute projections of participating relations. Since PERF join carries the *complete* join information, it is therefore capable of handling cyclic join queries without transmitting any useless tuples. In contrast, neither semijoin variants nor Bloomjoin can reduce the class of cyclic join queries, which does not have a full-reducer [BC81]. In addition, PERF join can handle inequality join while Bloomjoin will fail (due to the nature of hashing).

It is possible that due to a very high (or low) semijoin selectivity, PERF(R_i) will largely consist of 0's (or 1's). To reduce the network transmission cost, one can compress the encoded join information by sending the addresses of the 1 (or 0) bits within PERF(R_i) and perform a semijoin (or anti-semijoin [Gra93]) with R_i. Let P be the number of bits in PERF(R_i), K bits be the size of the PERF addresses, S be the number of 1's or 0's (whichever is smaller) then compression is preferred if $S < P/K$.

3.1 Improvements of PERF Join Over Semijoin

The "two-way" semijoin (2SJ) operator was introduced in [Dan82] and later promoted in [Seg86]. In [RK91], an improved version 2SJ+ was proposed. While 2SJ would send back to R the projection P'_S containing only the matching S join attribute values, 2SJ+ would ship the smaller of P'_S and $P_R - P'_S$ (containing non-matching values only) to R's site and perform either a semijoin or an anti-semijoin. Clearly, the backward transmission cost of 2SJ+ is bounded by half of P_R. When $|P'_S| > |P_R|/2$, 2SJ+ is more effective.

In [RK91] it is proved that the backward reduction of 2SJ+ is always cost-effective, i.e., that the benefits outweigh the costs, when the original site's relation needs to be transmitted to another site. We now prove that PERF join can, in principle, always do at least as well as 2SJ+, and often much better.

For simplicity of presentation, we assume both the forward and the backward semijoins are done at the sites storing the two joining relations, i.e., there is no third site involved. However it is easy to see that all the results would still hold when the semijoins take place at a different assembling site. As the forward reduction phase, reducing S by sending to it the join attribute projection of R, is the same for 2SJ, 2SJ+ and PERF join, we only compare their performance differences in their backward reduction phase.

Lemma 3.1: Let R and S be two joining relations, and suppose we wish to reduce both relations over their join attribute(s) X. Using the time it takes to reduce both relations as the comparison criterion, we have:
(1) The compressed PERF join is better than 2SJ+ if and only if

$$W(X) \quad > \quad \lceil \log(|R^X|) \rceil$$

[1]Note that S can represent several remote relations which will together reduce R. In addition, PERF can encode reduction information for other relational operators, such as intersection.

(2) The uncompressed PERF join is better than 2SJ+ if and only if

$$\frac{1}{W(X)} \quad < \quad \min(\rho(R), 1-\rho(R))$$

(3) The compressed PERF join is better than uncompressed PERF semijoin if and only if

$$\min(\rho(R), 1-\rho(R)) \quad < \quad \frac{1}{\lceil \log(|R^X|) \rceil}$$

Proof: Let $T = \lceil \log(|R^X|) \rceil$ represent the number of bits needed to index the relation R^X. We have two ways to implement PERF join: (a) send the PERF(R^X) without compression, with cost $|R^X|$ bits, or (b) compress the PERF and send the addresses of the 1-bits (or the 0-bits if there are fewer of them) with cost $T*M$ bits where M is the smaller number of matched or unmatched R^X values, e.g., $M = |R^X| * \min(\rho(R), 1-\rho(R))$.

The cost of 2SJ+ is $W(X) * M$ bits. The stated inequalities follow. ∎

Note that Condition (1) says that the compressed PERF join never does worse than 2SJ+ since $W(X) \geq \lceil \log(|R^X|) \rceil$ always holds.

Another important point to observe is that performing a PERF join might even be faster than executing two single semijoins *in parallel*. Since the backward reduction phase of PERF join is so cheap, it is likely that it could be performed in the time gap between the completion of the first and second semijoin executions.

Without loss of generality, let's assume $|R^X| < |S^X|$. The response time for the parallel single semijoins is $W(X) * |S^X|$. The response time for the uncompressed PERF join is $W(X) * |R^X| + |R^X|$ bits. Uncompressed PERF semijoin wins[2] if

$$1 + \frac{1}{W(X)} < \frac{|S^X|}{|R^X|}$$

which is likely unless S^X and R^X are extremely close in size.

With the small cost of the backward reduction in a PERF join, a forward semijoin that the query optimizer did not previously believe was profitable may now become profitable as part of a larger distributed query processing plan. This means that PERF join can render a wider range of profitable plan choices to the optimizer than 2SJ and 2SJ+.

Notice that both 2SJ and 2SJ+ require sending join attribute values to the receiving site and performing a semijoin with the receiving relation. If the receiving site's memory is limited, multiple passes over either the receiving relation or the received join attribute projection might be needed to perform the backward semijoin. In a fast network, this semijoin I/O cost can easily exceed the network transmission cost, as shown by [ML86]. In contrast, the backward PERF join can be done more cheaply. Due to the compactness of a bit vector representation, a PERF is likely to fit entirely in memory most of the time. Also recall that the bit order in the received PERF is the same as the tuple scan order used when R is scanned to generate P_R during the forward reduction phase. As long as R is scanned in the same tuple order during the backward reduction phase, we can "merge-join" R and PERF(R) based on this common tuple scan order. The I/O cost is one pass over R (and PERF(R) if it doesn't fit in memory).

The cost of PERF join is that the actual value of $P_{S'}$ needs to be reconstructed rather than being directly available at R's site. This means that we must remember how the initial R relation was scanned during the forward reduction phase (i.e., via a sequential table scan, via a sequential (clustered) or random (unclustered) index scan, etc.) and make sure to scan it in the same way during the backward reduction phase. We will elaborate on this issue in Section 5.

3.2 Bloomjoin and Its Limitations

We have introduced Bloomjoin in Section 1. Due to the compact size of Bloom filters, Bloomjoin can sometimes reduce the network and local join cost better than semijoin. It was demonstrated in [ML86] that distributed join algorithms employing Bloomjoin consistently performed better than those employing semijoins.

The two major drawbacks with Bloomjoin are (a) that only equality joins can be handled, and (b) that hash collisions result in a loss of join information. In contrast to a full reduction after semijoin, a relation will only be partially reduced after Bloomjoin.

There is another reason why non-joining tuples will remain in the reduced relation after a Bloomjoin. Previous work [WCS92] has shown that even for an n-way equality join $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ with the use of good hash functions, it is not always optimal to reduce R_i fully with Bloom filters generated from the other $n-1$ relations in order to minimize the overall query response time. The main reason is the significant latency involved to generate, transmit and join the Bloom filters with the receiving relations. This latency is further exacerbated in a heterogeneous environment. In such an environment, the remote sites can not always talk to each other [GL94, Tem94] since they may assume different network protocols, query languages and data models. To perform a Bloomjoin between two remote sites, a third site (usually the query site) has to act like a coordinator to relay the Bloom filter from the sending site to the receiving site. For instance, given $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, in order to reduce R_1 from say R_3, we need to send the Bloom filter of R_3 to the query site, relay it to and join with R_2, then send the Bloom filter of $R_2 \ltimes R_3$ to the query site again, relay the Bloom filter to R_1 and finally reduce R_1. Notice that executing a one-step Bloomjoin requires one scan of the join attributes of the sending relation, at least one scan of the entire receiving relation to perform the join reduction,[3] and two network hops to transmit the filter from the sending site to the receiving site through the query site. The latency quickly adds up for multi-step Bloomjoin reductions. And all this sequential processing makes it difficult to improve the query response time.

In conclusion, even in the optimal Bloomjoin plan for executing $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, some relation R_i may only be partially reduced through Bloom filters generated from a subset of the other $n-1$ relations. As a result, spurious tuples would be shipped to the

[2] Similar results hold for compressed PERF join.

[3] When a Bloom filter is too big to fit in memory, we might have to scan the receiving relation multiple times to complete the Bloomjoin.

assembling site, incurring extra network and join processing overhead. This is where PERF join can improve overall performance.

3.3 Improvements of PERF join Over Bloomjoin

After executing Bloomjoins among the remote relations with the best Bloom filter sequence, it is possible to apply PERF join among the partially reduced relations (if any) to further decrease the amount of spurious data being transmitted to the query assembling site, which in turn lowers the overall query response time. We refer to such strategy as Alg_P, and it is outlined below:

1. Apply Bloomjoin: Reduce each remote relation R_i with Bloom filters generated from a subset of the other joining relations. The resulting relation R_i' might still contain some non-joining tuples.
2. Send join attributes: Transmit join attribute projection(s) $P_{R_i'}$ of each R_i' in parallel to the assembling site. Join the $P_{R_i'}$'s and generate a PERF(R_i') for each corresponding R_i'.
3. Receive PERF: Ship PERF(R_i') to and combine with each R_i'. R_i' will be fully reduced into a relation R_i'' which contains only the matching tuples.
4. Transmit matching tuples: After excluding the join attributes,[4] send each R_i'' in parallel to the assembling site to compose the final join result.

Note that $\max_{1 \leq i \leq n}(|R_i''| * W(R_i))$ is a lower bound of query response time. If only Bloomjoin is employed in a distributed query processing algorithm, Step 1 will be executed first to hopefully eliminate most of the unmatched tuples in R_i, then R_i' will be sent directly to the assembling site. We refer to this strategy as Alg_B. We now quantify the difference in query response time between Alg_P and Alg_B.

Many previous researchers have looked at how to propagate Bloom filters in order to maximize the join reduction effect while trying to minimize query response time [Mul83, Mul90, ML86, WCS92]. We do not wish to extend their work here. Instead we simply take the best Bloom filter sequence that would have been generated by any of their algorithms, and "plug it in" to Step 1 of our Alg_P strategy.

Suppose in the optimal Bloomjoin plan, R_i is partially reduced by the Bloom filters of m out of the $n-1$ joining relations ($0 \leq m < n$), and sending the reduced R_i to the assembling site takes the longest network time. Let the aggregate semijoin selectivity of these m relations over R_i be ρ_i, and the other $n-1-m$ relations over R_i be $\overline{\rho_i}$. Assuming that semijoin reduction effects are independent, we can use the semijoin selectivity model proposed in [BGW+81][5] to derive the following:

$$\rho(R_i) = \rho_i * \overline{\rho_i} \quad (1)$$

To simplify the analysis, we assmue Step 3 uses the uncompressed PERF semijoin. Similar results hold if the compressed PERF join is used. Let the arrival time of the reduced R_i at the query site be RE_B for Alg_B and RE_P for Alg_P, respectively. Let the Bloom filter component of RE_B and RE_P be bf_m (the cost of Step 1). We then have:

$$\begin{aligned} RE_B &= bf_m + W(R_i) * |R_i| * \rho_i && (2) \\ RE_P &= bf_m + W(X_i) * |R_i| * \rho_i + |R_i| * \rho_i \\ &\quad +(W(R_i) - W(X_i)) * |R_i| * \rho(R_i) && (3) \end{aligned}$$

Alg_P has a smaller query response time if $RE_P < RE_B$. After plugging in (1) and simplifying (3) - (2), we get:

$$\overline{\rho_i} < 1 - \frac{1}{W(R_i) - W(X_i)} \quad (4)$$

For instance, given $W(R_i) - W(X_i) = 400$ bits, Alg_P is better than Alg_B if $\overline{\rho_i} < 0.9975$. In other words, if after applying the optimal Bloomjoin program, more than 0.25% of the reduced R_i are spurious tuples, then it is beneficial to apply an additional PERF join against the reduced R_i in order to further decrease the query response time. Based on our discussions in section 3.2, this is a highly likely scenario, especially when $W(R_i) - W(X_i)$ and n are large.

4 Performance Improvements

We now compare the performance of PERF join with other semijoin variants such as 2SJ, 2SJ+, parallel semijoins and Bloomjoin on some examples. Unless otherwise stated, we assume $R \bowtie S$ is considered.

Example 4.1: In this example, let's take $|R^X| = 2^{20}$, $W(X) = 80$ bits. The performance graph showing the network cost spent in the backward reduction phase using 2SJ, 2SJ+, compressed PERF join (CPERF) and uncompressed PERF join (UPERF) appears in Figure 2. We use log (base 2) scaling for the vertical axis in order to better view the performance differences between different methods. Note that the network costs of CPERF and UPERF are independent of W(X), and the cost of UPERF is independent of $\rho(R)$. The performance gaps will further widen given larger $|R^X|$ and $W(X)$ values.

The performance graph showing the response time using uncompressed PERF semijoin (UPERF) and parallel semijoins (PSJ) appears in Figure 3. Unlike in Figure 2, UPERF now counts both the forward and the backward cost. Note that the response time of UPERF is independent of $|S^X|$. The performance gaps will further widen given larger $|S^X|$ and $W(X)$ values.□

Example 4.2: In this example, we shall consider $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. Let's take $|R_i| = 2^{20}$, $W(R_i) = 432$ bits, $W(X_i) = 32$ bits, $bf_m = 2^{20}$, and $\rho_i = 0.1$. The performance graph showing the response time difference using Alg_P, Alg_B and the lower bound appears in Figure 4. Note that RE_B is independent of $\overline{\rho_i}$. The performance gap will further widen given larger $|R_i|$ and $W(R_i) - W(X_i)$ values. □

The following example shows PERF join can perform well when full Bloomjoin reduction is not possible. In particular, we can handle non-equality joins. Another situation where Bloomjoin does not apply is a join based on some notion of "similarity" rather than on strict equality. Examples include band joins [DNS91], and joins based on domain-specific similarity functions [HS95].

[4] The assembling site has already got a copy of the needed join attribute projections after Step 2.

[5] It is straightforward to generalize our results for a more complex semijoin selectivity model.

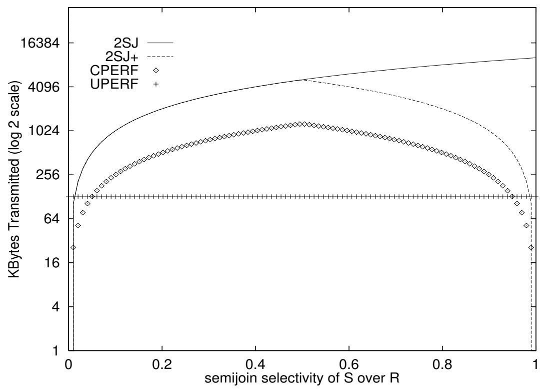

Figure 2: Response Time of PERF join vs. Semijoin Variants

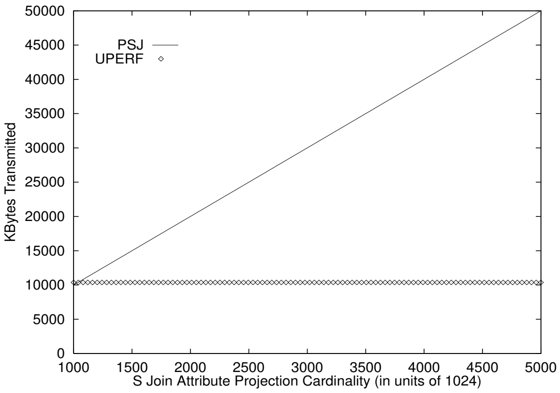

Figure 3: Response Time of Parallel 2SJ vs. PERF join

Example 4.3: Consider the example of a retailer X and a manufacturer Y who have their databases on-line for a distributor (the client) to read. The distributor wants to ask the query "Tell me all relevant information about products sold by X that are manufactured by Y, and for which X sold more than 5% of the total sold for that product." More specifically, let us assume that X has a relation R(*product-id*, *manufacturer*, *sales*), and that Y has relation M(*product-id*, *name*, *cost*, *total-sales*, *description*). (We abbreviate the attributes by their first letters.) The query can be phrased in relational algebra as

$$\sigma_{m=Y}(R \bowtie_{R.p=M.p \text{ and } s>0.05*t} M).$$

Imagine that X sells very many of Y's products, but very few of them qualify as more than 5% of total sales. Applying Bloomjoin could only reduce R and M based on the $R.p = M.p$ join condition, and would thus let through full tuples of mostly spurious matches.

Let $W(p) = W(s) = W(t) = 32$ bits, $W(R) = 320$ bits, $W(M) = 800$ bits, $|\sigma_{m=Y}R| = 4000$ and $|M| = 10000$. Let the semijoin selectivities of the equality condition over M and $\sigma_{m=Y}R$ be $\rho^E_M = 0.4$ and $\rho^E_R = 1.0$, respectively. Let $\rho(R) = 0.025$ and $\rho(M) = 0.01$. Let's assume that the number of bits in a Bloom filter is 1.2 times the relation cardinality.[6] Let $BF(Z)$ be the Bloom filter size of relation Z, we have $BF(\sigma_{m=Y}R) = 4800$ bits, and $BF(M) = 12000$ bits.

[6] This is still a conservative choice.

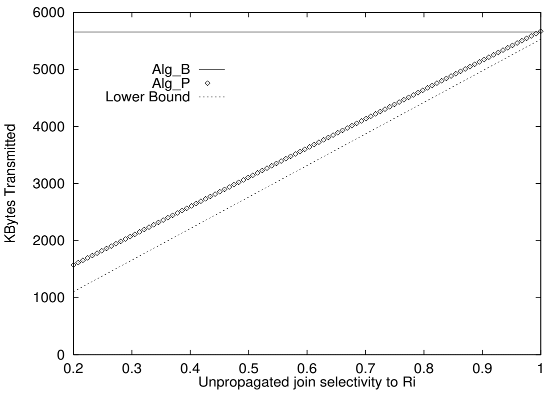

Figure 4: Response Time of Alg_P vs. Alg_B

If Bloomjoin is applied, one relation will receive the Bloom filter of the other. The partially reduced relation will be sent to the distributor site for the final join. The arrival time for R and M (in bits) are:

$$\begin{aligned} RE_R &= BF(M) + |\sigma_{m=Y}R| * W(R) * \rho^E_R \\ &= 1292000 \\ RE_M &= BF(\sigma_{m=Y}R) + |M| * W(M) * \rho^E_M \\ &= 3204800 \end{aligned}$$

If PERF join is applied, the relation (in this case it is R) having the smaller join attribute projection size will send the join attribute projection to the other and receive its PERF back. Finally the fully reduced relation will be sent to the distributor site for the final join. The arrival times for R and M (in bits) are:

$$\begin{aligned} RE_R &= (W(p) + W(s)) * |\sigma_{m=Y}R| \\ &\quad + \text{PERF}(|\sigma_{m=Y}R|) \\ &\quad + |\sigma_{m=Y}R| * W(R) * \rho(R) \\ &= 292000 \\ RE_M &= (W(p) + W(s)) * |\sigma_{m=Y}R| \\ &\quad + |M| * \rho(M) * W(M) \\ &= 336000 \end{aligned}$$

Therefore, the query response time after applying Bloomjoin and PERF join are 3204800 bits and 336000 bits, respectively, almost a factor of 10 difference. □

5 PERF Join Scan State Management

In order to process PERFs properly, it is necessary that the site receiving the PERF be able to reconstruct the order of tuples sent previously. The complexity of achieving this effect varies depending on what access path used to scan the relation tuples and whether duplicates are eliminated when performing the forward phase semijoin. In general, the following issues need to be addressed in order to guarantee the correctness of PERF join.

5.1 How to Maintain the Scan Order

Suppose that the PERF has been constructed on a remote relation R. If R is only updated periodically, a simple scheme which relies on some sort of locking can be used. Depending on how tuples of R are accessed, shared locks can be held on data pages or index pages. This isn't as bad as it sounds, as *any* distributed query processing algorithm has to handle this distributed concurrency control problem somehow, if consistency of the query result is required.

An alternative allowing better concurrency is to keep a version of R prior to sending any tuples. This can be done by maintaining the differential changes of the relation, or by simply materializing the version as a temporary relation.[7]

In either case, the access method used during the forward reduction phase needs to be recorded and kept for the backward reduction phase.

5.2 How to Handle Duplicate Elimination

In the forward semijoin reduction phase we need to eliminate duplicates in the join attribute projection. Otherwise we may transmit a significant amount of redundant information. Duplicate elimination is typically done by sorting. Thus there is the potential problem that the semijoin projection as transmitted may not be in the original scanning order of the relation. We address this problem below.

Suppose that R is the local relation, and X represents the projection of R onto the join attribute(s) without eliminating duplicates, ordered according to the scan ordering of R. (The scan ordering of R is either the physical order of R, or the order according to some index used for the scan.)

Add to X an extra column Y which is a sequence number in the projection, so that XY might look like

X	Y
a	1
b	2
a	3
a	4
c	5
b	6
a	7

Sort XY into X order, giving, for the example above

X	Y
a	1
a	3
a	4
a	7
b	2
b	6
c	5

Transmit just the X column to the remote site, without transmitting duplicates. When the PERF is received, make a pass through the sorted XY relation to mark all records with a nonmatching X value. This marking is achieved by making the corresponding Y value a number larger than all other Y values, say 99999. So, if the PERF was 101 for the above example ("abc" would have been initially transmitted) we would then have

X	Y
a	1
a	3
a	4
a	7
b	99999
b	99999
c	5

Now, re-sort the XY column on Y, to give

X	Y
a	1
a	3
a	4
c	5
a	7
b	99999
b	99999

The matching tuples are now in the original scan order, and we need only one pass through R to retrieve the tuples appearing in the final join result.

Since the XY relation is much narrower than R, and may have fewer tuples than R if there are local selection conditions, it is reasonable to assume that it will fit in memory, and so the sorting can be done without extra I/O. On the other hand, if the distributed query is going to take a long time, it is probably a bad idea to reserve memory for XY for the duration of the whole query, and so some extra I/O is probably needed to read and write XY between the forward and backward semijoins.

6 Conclusions

The main contribution of this paper is the proposal of a novel two-way join reduction implementation primitive called "PERF join", which is a competitive alternative to two-way semijoin and Bloomjoin. Through analytical studies, we have demonstrated that the response time of distributed join query processing algorithms can be improved by employing PERFs and PERF joins instead of or in addition to the traditional Bloomjoin and semijoin variants.

The following is a brief summary of the features of PERF-based techniques:

1. Preservation of complete join information.
2. Minimal network and storage overhead.
3. Cheap local join processing cost, especially when buffer memory is scarce.
4. Inequality join query handling.
5. Cyclic join query handling.

As future work, we are in the process of implementing the PERF join technique together with other optimizations in the context of a Parallel and Distributed Query Processing System (PDQPS) being developed at Columbia University [LR95]. We hope to demonstrate the *practical* utility of PERF join operators within this realistic general distributed query processing framework.

[7] It is actually more efficient to materialize R after applying the local selection and Bloomjoin (or semijoin) reduction. We leave out the details here due to space limitations.

References

[AHY83] P. Apers, A. Hevner, and S Yao. Optimization algorithm for distributed queries. *IEEE Trans. Software Eng.*, SE-9:57–68, 1983.

[Bab79] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Transactions on Database Systems*, 4(1):1–29, 1979.

[BC81] P.A. Bernstein and D.M. Chiu. Using semijoins to solve relational queries. *J.ACM*, 28(1):25–40, 1981.

[BG81] P.A. Bernstein and N. Goodman. The power of natural joins. *SIAM J. Computi.*, 10:751–771, 1981.

[BGW+81] Philip Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothie. Query processing in a system for distruibuted databases(sdd-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[CY93] Ming-Syan Chen and Philip S. Yu. Combining join and semi-join operations for distributed query procesing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):534–542, 1993.

[Dan82] D. Daniels. Query compilation in a distributed database system. IBM Res. Rep. RJ 3423, IBM, 1982.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th VLDB conference*, pages 443–452, 1991.

[GL94] Piyush Gupta and Eileen Lin. DataJoiner: A practical approach to multi-database access. In *Proceedings of Third International Conference on Parallel and distributed Information Systems*, page 264, 1994.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surverys*, 25(2):73–170, 1993.

[HS95] Mauricio Hernandez and Sal Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM-SIGMOD Conference*, 1995.

[HY79] A.R. Hevner and S.B. Yao. Query processing in distributed database system. *IEEE Trans. Software Eng.*, SE-5(3), 1979.

[LR94] Zhe Li and Kenneth A. Ross. Better semijoins using Tuple Bit-Vectors. Technical Report CUCS-010-94, Columbia University, New York, NY 10027, 1994.

[LR95] Zhe Li and Kenneth A. Ross. Federated query processing on the net using information servers. Submitted for publication, 1995.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th VLDB conference*, pages 300–311, 1986.

[Mul83] James K. Mullin. A second look at Bloom filters. *Communications of the ACM*, 26(8):570–571, 1983.

[Mul90] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[RK91] Nick Roussopoulos and Hyunchul Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Transactions on Knowledge And Data Engineering*, 3(4):486–495, 1991.

[Seg86] Arie Segev. Optimization of join operations in horizontally partitioned database systems. *ACM Transactions on Database Systems*, 11(1):48–80, 1986.

[Tem94] Marjorie Templeton. Interviso: A data integration server. In *Proceedings of Third International Conference on Parallel and distributed Information Systems*, pages 265–266, 1994.

[WCS92] Chihping Wang, Arbee L.P. Chen, and Shiow-Chen Shyu. A parallel execution method for minimizing distributed query response time. *IEEE Transactions on Parallel And Distributed Systems*, 3(3):325–333, 1992.

[YC84] C.T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surverys*, pages 399–433, 1984.