# Better Semijoins Using Tuple Bit-Vectors


| Zhe Li | Kenneth A. Ross |
| --- | --- |
| Computer Science Department | Computer Science Department |
| Columbia University | Columbia University |
| New York, NY 10027 | New York, NY 10027 |
| li@cs.columbia.edu | kar@cs.columbia.edu |



Technical Report No. CUCS-010-94
April 10, 1994



**Abstract**

This paper presents the idea of "tuple bit-vectors" for distributed query processing. Using tuple bit-vectors, a new two-way semijoin operator called 2SJ++ that enhances the semijoin with an essentially "free" backward reduction capability is proposed. We explore in detail the benefits and costs of 2SJ++ compared with other semijoin variants, and its effect on distributed query processing performance. We then focus on one particular distributed query processing algorithm, called the "one-shot" algorithm. We modify the one-shot algorithm by using 2SJ++ and demonstrate the improvements achieved in network transmission cost compared with the original one-shot technique. We use this improvement to demonstrate that equipped with the 2SJ++ technique, one can improve the performance of distributed query processing algorithms significantly without adding much complexity to the algorithms.

# Better Semijoins Using Tuple Bit-Vectors

Zhe Li Kenneth A. Ross*

Computer Science Department
Columbia University
New York, NY 10027
{li,kar}@cs.columbia.edu

**Abstract**

This paper presents the idea of "tuple bit-vectors" for distributed query processing. Using tuple bit-vectors, a new two-way semijoin operator called 2SJ++ that enhances the semijoin with an essentially "free" backward reduction capability is proposed. We explore in detail the benefits and costs of 2SJ++ compared with other semijoin variants, and its effect on distributed query processing performance. We then focus on one particular distributed query processing algorithm, called the "one-shot" algorithm. We modify the one-shot algorithm by using 2SJ++ and demonstrate the improvements achieved in network transmission cost compared with the original one-shot technique. We use this improvement to demonstrate that equipped with the 2SJ++ technique, one can improve the performance of distributed query processing algorithms significantly without adding much complexity to the algorithms.

## 1 Introduction

With present day technology, the main cost of query processing in distributed databases is the overhead of network communication, particularly over wide-area networks. Thus, distributed query processing strategies should attempt to minimize the amount of data transmitted over the network. A popular technique for reducing network transmission volume is the use of *semijoins*.

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

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

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.

Semijoins are one possible step in a distributed query processing algorithm. Much work had been done on optimizing the cost of distributed query processing using various cost models [ESW78, HY79, AHY83, Won77, YC84, RK91, WCS92, CY93]. Most of them can be classified as semijoin based, and a distributed query is typically processed 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 remote joins between different sites. A cost-effective semijoin program is then derived and executed (usually in sequential steps, but possibly in parallel) to reduce the size of the relations involved. Often both the local processing cost and network transmission cost can be reduced significantly.

3. Final assembly: All the reduced relations that are needed to compute the final result are shipped to a final site where the result is assembled.

An important extension of the semijoin operation is the "two-way" semijoin operation, which we abbreviate as 2SJ. 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 semijoins are particularly useful when both relations $R$ 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 extended 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+.

This paper presents the idea of Tuple Bit-Vectors and a new two-way semijoin operator called 2SJ++ which enhances the semijoin with an essentially "free" backward reduction capability. The basic idea is as follows: Instead of transmitting $P_{S'}$ back to $R$, send a bit-vector 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 that $R$'s site sent initially. Assuming that $R$'s site can easily reconstruct this order, the value of $P_{S'}$ can be reconstructed at $R$'s site from the bit vector.

In many cases the bit vector 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 bit vector is only 12500 bytes. The tuple bit vector 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.

We explore in detail the pros and cons of 2SJ++ compared with other semijoin variants. We then use the "one-shot" parallel distributed query processing algorithm [WCS92] as an application of 2SJ++. We use 2SJ++ within the one-shot algorithm and demonstrate the improvements achieved in the network transmission cost. We use this application to demonstrate that equipped with the conceptually simple yet generally applicable 2SJ++ idea, we can improve the performance of distributed query processing algorithms significantly without adding much complexity to the algorithms.

The rest of the paper is organized as follows. Section 2 gives the terminology and assumptions adopted throughout this paper. In Section 3 we introduce the notion of tuple bit-vectors and use them to define a two-way semijoin operator "2SJ++". Section 4 gives a brief overview of the "one-shot" algorithm. In Section 5 we describe the performance improvements that can be achieved by incorporating 2SJ++ into the one-shot algorithm. In Section 6 we conclude and describe further research problems.

# 2 Terminology and Assumptions

We assume that we have $n$ relations $R_1, \ldots, R_n$, located at $n$ distinct sites $1, \cdots, n$. The query that we are trying to answer is of the form

$$\pi_A \sigma_C (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$$

where $A$ is a set of attributes, and $C$ is a condition on the attributes of $R_1, \ldots, R_n$.

A *distributed query processing algorithm* is defined to be a set of relational operations and network transmission steps, such that, at the end of executing the algorithm, the $n$-way join result is computed and present at the query originating site.

We list below some of the notations we use throughout this paper:

| | |
|---|---|
| $R_i \ltimes R_j$ | semijoin. |
| $\|R\|$ | cardinality of $R$. |
| $X_{(i,j)}$ | join attribute(s) common to $R_i$ and $R_j$. |
| $R^X$ | $\pi_X(R)$, removing duplicates. |
| $\rho_j^i$ | join/semijoin selectivity of $R_j$ with respect to $R_i$. |
| $W(R)$ | width of a tuple in $R$ (in bytes). |
| $V(X)$ | width of the join attribute(s) $X$ (in bytes). |
| $D_j$ | per byte constant cost of network transmission from site $j$. |
| $E_j$ | network setup time at site $j$. |
| $F$ | the coefficient of the most significant term in the join cost. |
| $C_i$ | the scanning cost of $R_i$ at site $i$. |

Logarithms are all base 2.

Throughout the paper we make the following assumptions:

- Attribute values are distributed uniformly.
- The cost of applying a hash function is negligible.
- The network cost is measured as the number of bytes transmitted.

# 3 Tuple Bit Vectors

In this section we describe several semijoin techniques and extend them using the tuple bit-vector idea.

## 3.1 Hash Filters

Hash filters were introduced in [Blo70, Bab79] and promoted later in [Mul83, Mul90, Qad88]. A hash filter is a bit vector used to encode the joining relationship. When joining $R_i$ with $R_j$, the join attribute values of $R_i$ are hashed to some addresses in the bit vector whose corresponding bits are then set to 1. A zero bit after hashing would indicate that no attribute value that hashes to that bit participates in the join. A significant reduction in network cost can often be obtained compared with sending the actual semijoin projection values. For instance, let $R$ be a relation, and let $P_R$ be the projection of $R$ onto its join attributes. Suppose that the join attributes have total size 4 bytes, and that the size of the hash vector is four times the cardinality of $P_R$. We end up with sending $4 * |P_R|/8 = |P_R|/2$ bytes of a hash filter compared with sending $4 * |P_R|$ bytes of semijoin values, a saving by a factor of 8.

The two major drawbacks with the hash filter based approach are (a) that only equality join can be handled, and (b) the hash collisions result in a loss of join information, so the actual reduction effect would be worse than joining with the actual attribute values.

The problem of how to choose the size of a hash filter to effectively control hash collisions is not well addressed in previous research. Suppose we wish to compute $R_j \ltimes_X R_i$ and assume that $|R_i| \ll |R_j|$, and that $\rho_i^j$ is very small. Even in the presence of a perfect hash function $f$ (that distributes join keys uniformly), if the hash filter size $H$ is small compared with $|R_j|$, hash collisions would occur frequently and the corresponding semijoin reduction effect on $R_j$ would be compromised by a large factor. The following example illustrates the problem in more detail:

**Example 3.1:** Let $|R_i| = 10^4$, $|R_j| = 10^9$, $H = 10^h$. Assume $R_i.X$ and $R_j.X$ are key attributes in both relations and $R_i.X \subset R_j.X$. (Thus, duplicates play no role in this example.) If we use a standard value-based semijoin, the number of unmatched $R_j$ tuples would be $10^9 - 10^4$, the result size $|R_j \ltimes_X R_i|$ would be $10^4$ and the selectivity $\rho_i^j = 10^{-5}$.

Now assume a hash function $f$ is applied to $R_i.X$ and the hash filter is sent to $R_j$ to perform a semijoin. After hashing on $R_i.X$, there are (assuming few collisions) approximately $10^4$ bits set in the hash filter. When we hash on $R_j.X$, the probability that each unmatched tuple in $R_j$ hashes to a bit set by $R_i$ is $10^{4-h}$. Thus the expected number of spurious hash matches in $R_j$ is:

$$E(\text{spurious matches}) = (10^9 - 10^4) * 10^{4-h} \approx 10^{13-h}$$

.

The net reduction factor is:

$$\frac{10^4 + 10^{13-h}}{10^9} = 10^{-5} + 10^{4-h}$$

For instance, if $h = 7$, the reduction effect on $R_j$ using hash filter based approach would be two orders of magnitude smaller compared with that achieved by using value-based semijoin. Even with $h = 9$, half of the hash matches are spurious tuples, effectively doubling the number of tuples of $R_j$ that are later transmitted.

With $h \geq 11$, the number of spurious tuples drops to about 1%. In this case, even using a compressed representation of the hash table (sending 36-bit table addresses, see Section 3.6) the amount of data transmitted would be about 45000 bytes. If the size of the attribute $X$

were 4 bytes, then sending the values would require only 40000 bytes, and yield no spurious tuples. □

To summarize, the size of a hash filter may need to be a lot bigger than the larger of the joining relations to avoid a severe collision problem. As a result, the network cost of sending a hash filter may be higher than simply sending the actual join values. Additionally, unmatched tuples will be unnecessarily sent later to the assembling site, which adds extra network and local-processing delays to the overall query processing time.

Chosen appropriately however, semijoins based on hash filters can yield orders of magnitude reduction in network overhead and lower semijoin cost at the receiving site. If the hash filter can fit into the receiving site's memory, we need only scan the receiving relation once to perform hashing and produce the semijoin result.

## 3.2 Two-Way Semijoins

The two-way semijoin (henceforth referred to as 2SJ) was introduced in [Dan82] and later promoted in [Seg86]. It is usually implemented as follows: for $R_i \bowtie R_j$ with join attribute(s) $X$, we (a) send $R_j^X$ to site $i$, (b) perform $R_i \ltimes R_j^X$ yielding a new relation $P$, then (c) send back to site $j$ the reduced $P^X$ for a restriction of $R_j$.

In [RK91], an improvement of 2SJ was proposed. We denote the new operator by "2SJ+". It is implemented by modifying step (c) above to send back the smaller of $P^X$ and $R_j^X - P^X$. Clearly, the additional transmission cost of 2SJ+ over the semijoin is bounded by half of $|R_j^X|$. In contrast, 2SJ always sends $|P^X|$ tuples during the backward reduction phase. When $|P^X| > |R_j^X|/2$, 2SJ+ is more effective.

Lemma 2 in [RK91] shows that if the original semijoin is cost-effective, in a formally defined sense, then the 2SJ+ is also cost-effective. Moreover, the *backward reduction* is always cost-effective. In other words, *it always pays to do the backward reduction if the initial relation is going to be transmitted to another site.*

In the next section we show how we can reduce the cost of the backward reduction even further.

## 3.3 Improving 2SJ+ with Tuple Bit-Vectors

We present a further improvement of 2SJ+ [RK91].

**Definition 3.1:** Let $R$ be a relation whose tuples are ordered in some fashion. A *tuple bit vector* $V_R$ of relation $R$ is an array of $|R|$ bits. The $i$th bit of the array corresponds to the $i$th tuple of $R$.□

We now explain how tuple-bit vectors can speed-up the backward reduction phase of 2SJ+. Suppose we construct a tuple bit vector $V = V_{R_j^X}$ and place a 1 in $V$ in the bit position of every tuple in $P^X$ (as defined above). $V$ encodes $P^X$; $P^X$ can be reconstructed at site $j$ given $V$ and the original order of $R_j^X$.

The tuple bit-vector size $|V_R|$ of relation $R$ is bounded by $|R|$. Thus, if $|P^X| \approx |R_j^X|/2$, we send $|R_j^X|$ bits instead of $16 * |R_j^X|$ bits (assuming that attribute $X$ occupies 4 bytes). Even if $|P^X| \ll |R_j^X|$ we can use a compression scheme to transmit $|P^X| \log(|R_j^X|)$ bits rather than $32 * |P^X|$ bits (see Section 3.6).

Since the size of the tuple bit-vector could be orders of magnitude smaller than the size of the semijoin projection, and because the semijoin with a tuple bit-vector can be implemented by a one-pass scan of the receiving relation, tuple bit-vectors make the backward reduction phase of 2SJ++ essentially "free" compared with the cost of forward reduction.

The cost of 2SJ++ is that the actual value of $P^X$ needs to be reconstructed rather than being directly available. This means that we must remember how the initial relation was scanned (i.e., the whole relation sequentially, via an index, a sequential range within the relation, etc.) and make sure to scan it in the same way. Storing the access method used should use very little space. Note that only one pass of the data is needed to apply the "backward" semijoin; encoding the backward semijoin as a tuple bit vector does not necessitate additional passes over the initial relation during the local processing phase.

## 3.4 Improving Hash Filters with Tuple Bit-Vectors

Similar techniques can be used in a backward reduction phase for hash filters. Imagine a hash table $N$ bits long that has $M$ bits set. Instead of returning a hash table $H$ that is $N$ bits long, we return a table $T$ that is $M$ bits long. The $i$th bit of $T$ is set if the $i$th 1-bit of the original hash table is set. Given the original hash table and $T$, the table $H$ can be reconstructed. The savings will be particularly high if the table $H$ is sparse. (As illustrated in Example 3.1 we may try to generate a sparse table to minimize collisions.)

Thus we shall also consider a two way semijoin based on hash filters that uses tuple bit vectors for the backward reduction as an example of 2SJ++. Note that 2SJ+ does not apply in this case.

## 3.5 Comparing 2SJ++ with 2SJ+ and Parallel Semijoins

In the following discussion we measure the cost and benefit of a semijoin as the network cost, i.e., the number of bytes transmitted and reduced.

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

**Lemma 3.1:** Let $R_i$ and $R_j$ be two relations, and suppose we wish to perform a two-way semijoin of these relations over join attribute(s) $X$. Let $V$ be the size (in bytes) of attribute(s) $X$. Then the cost of 2SJ++ is less than or equal to the cost of 2SJ+ if $8 * V \geq \log(|R_j^X|)$.
*Proof*: Let $X$ be the join attribute(s), let $N = |R_j^X|$ be the cardinality of $R_j^X$, and let $S = \lceil \log(N) \rceil$ represent the number of bits needed to index the relation $R_j^X$. We have two options in 2SJ++ if we allow compression: (a) send the tuple bit vector for the whole table, with cost $N/8$, or (b) send the addresses of the 1-bits (or the 0-bits if there are fewer of them) with cost $\frac{S*M}{8}$ where $M$ is the number of matched (respectively, unmatched) tuples.

We choose the smaller of these two costs. The cost of 2SJ+ is $V * M$. The cost of 2SJ++ is at most $\frac{S*M}{8}$. Thus 2SJ++ is cheaper if $8 * V \geq S$, and the result is proved. ▮

When combined with results from [RK91], Lemma 3.1 states that the extension of the semijoin to the 2SJ++ semijoin is always done in a cost-effective way if $8*V \geq S$. For every single attribute value or table address transmitted by the 2SJ++ in the backward reduction, at least one whole tuple in the $R_j$ relation is eliminated. Note that the tuple bit-vector scheme used doesn't lose join information.

If $8 * V$ is smaller than $S$ in Lemma 3.1 above then it may be preferable to send the join values directly if $8 * V * M \leq |R_j^X|$. All of the parameters necessary for this estimation are available in order to calculate the best strategy. Thus we can augment 2SJ++ with a test for the above condition, and revert to 2SJ+ in that case. However, we should remark that it would require an extreme circumstance for $8 * V \leq S$ to hold: if $V = 4$ then we need at least $2^{32}$ tuples in the original transmitted projection.

On the other hand, the saving may be very large. For example, if $V = 4$, $N = 2^{20}$, $M = 2^{18}$, then the reduction in the cost of the backward-reduction phase is at least $\frac{N}{8*V*M} = \frac{1}{8}$.

Note that it is possible that with the reduced cost of the backward reduction, a semijoin that the optimizer did not previously believe was profitable may now become profitable as part of a larger distributed query processing plan.

Another important point to observe is that performing a two-way semijoin using 2SJ++ may be faster than doing two single semijoins *in parallel*. Since the backward reduction phase is so cheap, it is relatively likely that it could be performed in the time gap between the completion of the first and second semijoin reductions.

To show that the gap between the two parallel single semijoins is large with respect to the size of the backward reduction, we calculate below the difference in the network transmission time in both directions. Assume that $D_i = D_j$, and that, without loss of generality, $|R_j^X| \geq |R_i^X|$. The difference in response time for the parallel single semijoins is approximately $D_i * V * (|R_j^X| - |R_i^X|)$ where $V$ is the size of the join attribute(s) $X$. The transmission cost of the backward-reduction phase is $D_i * |R_i^X|/8$ which is less than the difference above if

$$1 + \frac{1}{8 * V} \leq \frac{|R_j^X|}{|R_i^X|}$$

which is likely unless $R_i^X$ and $R_j^X$ are extremely close in size.

## 3.6 Compression of Hash Tables and Tuple Bit-Vectors

In [Qad88], two compression techniques were proposed to further reduce the network cost of sending hash filters. They are based on the assumption that hash filters are usually very sparse, that is, they contain a small number of bits with the value "1" scattered among very many bits having the value of "0". This is true if the number of bits in the hash vector is a large multiple of the relation size (as in Example 3.1, or if there is a high degree of duplication in the join column).

The two compression schemes are as follows:

**Scheme 1** : In this scheme, instead of sending all the bits in the hash filter, only the hash addresses of those bits in that filter which have the value "1" are sent. A site receiving the bit hash addresses could use them to set up its own hash filter.

**Scheme 2** : The number of zero's between two adjacent nonzero bits in the bit vector is converted into a binary number of length $4, 8, 12, \ldots$ bits. The code words generated by a node are collected into pages. The set of code pages generated by the sending site is transmitted across the network to the receiving site. The receiving site then uses the received code words to set up the hash filter.

The experimental data obtained in [Qad88] suggests a substantial improvement in the performance of a hybrid-hash-join algorithm implemented using hash filters after taking advantage of these two compression schemes. In addition, scheme 1 is suggested to be the best choice.

If $H$ is the size of the hash table (in bits), then it takes $\lceil \log(H) \rceil$ bits to encode an address in the table. If there are $N$ 1-bits in the table, we prefer to send addresses rather than the whole table if $N * \lceil \log(H) \rceil \leq H$, i.e., if

$$N \leq \frac{H}{\lceil \log(H) \rceil}$$

# 4 Overview of the "One-Shot" Algorithm

The "one-shot" algorithm was proposed in [WCS92]. As opposed to the traditional sequential semijoin paradigm [HY79], this method executes all applicable semijoins to the relations *in parallel.* That is, each relation will be reduced by a set of semijoins at one time, i.e, in "one shot," and the semijoin processing at all sites can be performed simultaneously. As a result, each relation needs to be scanned only once to process all applicable semijoins. The One-Shot algorithm uses hash-filters for its semijoins.

The aim of the one-shot algorithm is to optimize *response time* rather than the amount of *work done.* Thus, some extra work in the form of additional (parallel) data transmission is tolerated in order to improve the overall response time.

The query optimizer chooses a semijoin program which is a set (instead of a sequence) of semijoins for each relation. For each site $j$, a set $B_j$ of other sites is chosen. Site $i \in B_j$ if we want to apply the semijoin $R_j \ltimes R_i$ to $R_j$. The goal is to find a set $\{B_1, B_2, \ldots, B_n\}$ that minimizes the overall response time.

Let $s_i^j$ be the total time needed to project $R_i$, hash on the join attribute values, and transmit the hash filter to the site where $R_j$ is located. The cost model of "One-Shot" method can be expressed as follows, using the terminology of Section 2. The response time $RESP(B_1, B_2, \ldots, B_n)$ is given by

$$\max_{1 \leq j \leq n} \left( \max_{i \in B_j} s_i^j + C_j + E_j + D_j * |R_j| * \prod_{i \in B_j} \rho_i^j \right) + F * \prod_{j=1}^{n} (|R_j| * \prod_{i \in B_j} \rho_i^j) \qquad (1)$$

Given an appropriate database profile, including selectivity statistics, etc., the response time above can be estimated. The authors of [WCS92] made the following two observations to reduce the search space for optimal query processing strategies from exponential to polynomial:

1. If $B_j$ is optimal and $i \in B_j$, then $\{h | s_h^j < s_i^j\} \subset B_j$. Intuitively speaking, if we're sending a semijoin projection from site $i$ to site $j$, and the semijoin projection from site $h$ would arrive at site $j$ before the information from site $i$, then we may as well send the projection from site $h$. The projection from site $h$ can further reduce the size of $R_j$ without introducing an extra delay in response time (assuming network availability).

2. Let $v_j$ denote the subterm of Formula 1 inside the outer maximum. Let m be the value of $j$ such that $v_m$ achieves the maximum in Formula 1. In other words, $R_m$ is the last

relation to arrive at the final assembly site. Therefore, for any relation $R_h$ (including $R_m$), $B_h$ must be chosen such that $v_h \leq v_m$. However, among all $B_h$'s which satisfy this requirement, the optimal strategy has to pick the one with the smallest selectivity because the final size of $R_h$ can thus be most reduced without incurring extra delay to the overall response time.

For readers interested in the correctness of the cost model and the details of the optimization algorithm, we refer to [WCS92] for further details.

One potential problem with the "one-shot" algorithm is that each participating relation is not always fully reduced by the semijoin program. One reason is because $B_j$ is not necessarily a complete set of sites whose relations share a join attribute with the relation at site $j$. The other reason is hash collisions caused by using hash filters. We may end up transmitting more data to the final assembly site. This way we end up with additional I/O costs incurred by this incomplete filtering during the final assembly join phase, which contributes a significant cost factor to the total response time.

Another potential problem with the One-Shot Algorithm is that it does not take advantage of the relative efficiency of a two-way semijoin compared with the parallel execution of two one-way semijoins.

# 5 Improvements to "One-Shot" Algorithm

In principle, 2SJ++ can be used to enhance any parallel execution of distributed queries. In this section, we show how one such algorithm can be extended.

## 5.1 2SJ++ Based "One-Shot" Algorithm

As we have seen in Example 3.1, hash filters may not always be the best way to implement a semijoin. An initial step in the improvement of the one-shot algorithm would be to use database statistics to estimate when sending the attribute values would be preferable to sending the hash filters. Since hash filters can be used only for equality joins, sending attribute values may be necessary if the joins are not equality joins. For different semijoins in the semijoin program there may be different choices.

For either choice, i.e., hash filters or attribute values, the idea of sending tuple bit vectors for the backward reduction phase still applies.

There are several ways that using 2SJ++ rather than a sequence of single semijoins could either improve the response time or reduce the total amount of network bandwidth used. The first observation is that the one-shot algorithm may choose to have $i \in B_j$ and $j \in B_i$, performing the two single semijoins in parallel. As discussed in Section 3.5, the two-way 2SJ++ semijoin is likely to be preferable.

Using the two-way semijoin rather than two single parallel semijoins has two benefits. Let us assume that site $i$ receives the projection from site $j$ before site $j$ receives the projection from site $i$; in other words, $i$ finishes first. Then the backward reduction from site $i$ to site $j$ could be performed resulting (usually) in site $j$ finishing sooner. Thus, if site $j$ is the slowest site, the response time for the whole query will be reduced. This is the first benefit. The second benefit is that it uses fewer network resources to transmit the backward reduction, and so the total amount of work done is smaller.

The next observation is that the optimizer can know that 2SJ++ is being used, and can predict the smaller cost of the two-way semijoin (compared with two one-way semijoins). This extended optimizer may choose to perform a two-way semijoin when it would have only performed a one-way semijoin originally. The semijoin in the opposite direction may have been too expensive. In fact, since we have observed that the cost of the backward reduction is insignificant compared with the cost of the forward reduction, it is likely that the optimizer would choose to use 2SJ++ *for every semijoin operation.*

With this extra option for optimization, the reduction effect can only be improved in the semijoin phase of one-shot algorithm with the semijoin phase taking less time. Thus, the overall response time may be dramatically reduced.

To illustrate the potential gains, we present an example below. In this example, we count only the network cost, to simplify the discussion. The same principle holds when local processing cost is taken into account; a more detailed example will be presented in the full version of this paper.

**Example 5.1:** Suppose we have three sites, 1, 2, and 3. The result of $R_1 \bowtie R_2$ is required at site 3. Let $X$ be the join attribute. Let $|R_1| = 2^{13} (\approx 10^4)$, $|R_2| = 2^{20} (\approx 10^6)$, and assume $R_1.X$ and $R_2.X$ are key attributes in both relations. (Thus, duplicates play no role in this example.) Suppose that there are $2^{12}$ matching pairs of tuples in $R_1$ and $R_2$. Suppose we use hash filters for the semijoin with a hash table size of $2^{24}$ bits and assume (for simplicity) that there are no hash collisions. Finally, suppose that the size of a tuple in $R_1$ is $x$ bytes, (say $x = 2^8$) and the size of a tuple in $R_2$ is $y$ bytes (say $y = 2^8$). Since the hash table has size $2^{24}$, it requires 24 bits (or 3 bytes) to represent a hash address.

Let the per-byte network cost between any pair of sites be one unit in an appropriate scale. (We assume, for simplicity of presentation, that the network cost is uniform.)

The One-Shot algorithm would consider the following four cases:

1. $B_1 = B_2 = \emptyset$, i.e., no semijoin reduction. The total network cost is $x * 2^{13} + y * 2^{20} \approx 2^{28}$. The response time is $\max(x * 2^{13}, y * 2^{20}) \approx 2^{28}$.

2. $B_1 = \{2\}, B_2 = \emptyset$, i.e., a semijoin from site 2 to site 1. The total network cost is $2^{24}/8 + x * 2^{12} + y * 2^{20} \approx 2^{28}$. The response time is $\max(2^{24}/8 + x * 2^{12}, y * 2^{20}) \approx 2^{28}$.

3. $B_1 = \emptyset, B_2 = \{1\}$, i.e., a semijoin from site 1 to site 2. Applying the compression scheme of sending hash addresses to site 2 instead of the bigger hash filter, the total network cost is $3 * 2^{13} + x * 2^{13} + y * 2^{12} \approx 3 * 2^{20}$. The response time is $\max(x * 2^{13}, 3 * 2^{13} + y * 2^{12}) \approx 2^{21}$.

4. $B_1 = \{2\}, B_2 = \{1\}$, i.e., two parallel semijoins. The total network cost is $3 * 2^{13} + 2^{24}/8 + x * 2^{12} + y * 2^{12} \approx 2^{22}$. The response time is $\max(2^{24}/8 + x * 2^{12}, 3 * 2^{13} + y * 2^{12}) \approx 3 * 2^{20}$.

The one-shot algorithm equipped with a two-way 2SJ++ semijoin using hash filters, starting at site 1 has the following costs:

**Total network cost** $3 * 2^{13} + 2^{13}/8 + x * 2^{12} + y * 2^{12} \approx 2^{21}$.

**Response time** $\max(3 * 2^{13} + y * 2^{12}, 3 * 2^{13} + 2^{13}/8 + x * 2^{12}) \approx 2^{20}$.

Thus, in this case, we achieve an improvement by a factor of 1.5 in the total network cost and an improvement by a factor of 2 in (the network component of) response time. Note that the One-Shot algorithm would not have chosen to perform two semijoins,[1] since that option (number 4 above) performs worse than the single semijoin option (number 3). Nevertheless, the two-way semijoin using 2SJ++ proved better than all of the One-Shot plans. □

The improvement of 2SJ++ over 2SJ+ is apparent when (a) using semijoin values rather than hash filters, since 2SJ+ does not apply in the hash-filter case, and when (b) $x$ and $y$ (as in Example 5.1) are relatively small. In that case, the cost of the semijoin is a larger proportion of the total cost, and the saving can be significant.

# 6 Conclusions

The main contribution of this paper is the idea of "Tuple Bit-Vector". A new two-way semijoin operator called 2SJ++ is designed which enhances the semijoin with an essentially "free" backward reduction capability.

We compare 2SJ++ with other semijoin variants, and analyze its effect on distributed query processing performance. Using the "one-shot" algorithm [WCS92] as an application, we show that 2SJ++ could be used to enhance the performance of distributed query processing strategies. These observations and ideas are, in principle, also applicable to other distributed query processing algorithms, and could be easily implemented within them.

In future work, we plan to implement the 2SJ++ techniques in the context of a distributed relational query processing system being developed at Columbia University. We hope to demonstrate the *practical* utility of 2SJ++, together with other optimizations, within a realistic general distributed query processing framework.

# References


[AHY83] P.M.G. Apers, A.R. Hevner, and S.B. 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 semi-joins 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.

[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.


[1] Considering only the network cost.

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

[ESW78] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 169–180, 1978.

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

[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.

[Qad88] Ghassan Z. Qadah. Filter-based join algorithms on uniprocessor and distributed-memory multiprocessor database machines. In *Lecture Notes in Compure Science 303*, pages 388–413, 1988.

[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.

[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.

[Won77] E. Wong. Retrieving dispersed data from sdd-1: A system for distributed databases. In *Proceedings of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, 1977.

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