A New Client-Server Architecture for Distributed Query Processing

Zhe Li
Computer Science Department
Columbia University
New York, NY 10027
li@cs.columbia.edu

Kenneth A. Ross
Computer Science Department
Columbia University
New York, NY 10027
kar@cs.columbia.edu

Technical Report No. CUCS-014-94
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.

A New Client-Server Architecture for Distributed Query Processing

Zhe Li   Kenneth A. Ross*

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

Abstract

This paper presents a new client-server architecture to process distributed queries. We transform an N-way join query into an N-way join operation on the join attribute values only, and a concatenation operation to assemble the final join result. Rather than relying on conventional semijoin techniques, we use tuple bit vectors to effectively reduce the network overhead. We demonstrate that under this new architecture, the autonomy of the remote sites involved in executing the distributed query is preserved to a maximum degree, no direct communication link or network traffic among the remote sites is necessary, the available access paths on the participating relations can be effectively utilized and each joining relation is usually scanned only twice. The bulk of the processing is done by the client, allowing better utilization of the server. Several techniques are proposed to handle the potential processing bottlenecks that might occur before and during the final assembling stage. Finally we explore some of the query optimization and performance issues encountered when following this architecture to process distributed N-way join queries.

1 Introduction

Much work had been done on optimizing the cost of distributed query processing (henceforth abbreviated as DQP) under various cost models [ESW78, HY79, AHY83, Won77, YC84, RK91, WCS92, CY93]. Most DQP algorithms rely on some variants of the semijoin technique [BC81, BG81], and concentrate on reducing the cost of inter-site data movement. A distributed query is typically processed in the following stages [YC84]:

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

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

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.

   Note that no indexes will be shipped with the reduced relations to the final assembly site. It is likely that the join operation would be executed using the more costly versions of join algorithms [SALP79]. In the case of a low join selectivity, i.e., the query result is a near cartesian product, the final assembly join operation would be very expensive, representing a dominant cost factor in the overall query processing time.

There are a number of disadvantages for this architecture. First, the optimality of the query plans generated by the corresponding algorithms all depend on the accuracy of semijoin selectivity and intermediate join result size estimations. The semijoin selectivity estimation techniques adopted in the past are simply inherited or derived from the model in [Yao77, BGW+81], which makes simplistic assumptions about the data such as uniform distribution of values. Due to the severe estimation error propagation effects [IC91], the optimal strategies produced are sometimes no superior than a randomly chosen plan. Second, this architecture also makes the assumption that the optimizer can dispatch the workloads to any participating sites, e.g., site autonomy is not preserved at all. This is not necessarily the case in a real world setting, as it is perfectly legitimate that a remote server is only willing to provide the service of fetching requested data through filtered table scans, rather than providing the computing resources to service the join requests of external users.

This paper presents a new client-server architecture to process distributed queries. We transform an N-way join query into an N-way join operation on the join attribute values only, and a concatenation operation to assemble the final join result. Rather than relying on the conventional semijoin techniques which are sensitive to inaccurate database statistics, we use tuple bit vectors to effectively reduce the network overhead. We demonstrate that under this new architecture, the autonomy of the remote sites involved in executing the distributed query is preserved to a maximum degree, no direct communication link or even network traffic among the remote sites is necessary, the available access paths on the participating relations can be effectively utilized, and each joining relation is usually scanned only twice. Several techniques are proposed to handle the potential processing bottlenecks that might occur before and during the final assembling stage. Finally we explore some of the query optimization and performance issues encountered when following this architecture to process distributed N-way join queries.

The rest of the paper is organized as follows. Section 1.1 gives the terminology and assumptions adopted throughout this paper. In Section 2 we introduce the notion of tuple bit-vectors and present their advantages over other semijoin variants. Section 3 describes the new distributed query processing architecture and its advantages. In section 4 techniques are presented to effectively parallelize join operations and handle the potential bottlenecks that might occur at various processing stages. In Section 5 we give an example to show the performance improvements that can be achieved following this architecture compared with one of the parallel and distributed query processing algorithm. Section 6 surveys related work. In Section 7 we conclude and describe further research problems.

1.1 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.

A *tuple connector* is defined as as a projection of relation R on all the joining attributes and the corresponding tuple identifiers (TID) [RK91]. Each row in the tuple connector uniquely identifies a whole tuple in the corresponding relation.

The *canonical representation* of a join is a set of tuples $(TID_{R_1}, \ldots, TID_{R_n})$ containing the tuple identifiers of the matching tuples in the component relations. The *join result tuple connector* contains tuples that have both the tuple identifiers of the matching tuples together with the actual values of the join attributes that are present in the result relation.

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

$R_i \ltimes R_j$	semijoin.
$\pi_X(R)$	The projection of R onto X without removing duplicates.
$\|R\|$	cardinality of R.
$W(X)$	width of the attribute(s) X (in bytes).
V_i	the tuple bit vector for R_i.

2 Tuple Bit Vectors

In this section we describe the use of tuple-bit-vectors, as proposed in [LR94].

2.1 Improving Two-Way Semijoins with Tuple Bit-Vectors

A one-way semijoin aims to reduce a relation R to contain only those tuples that match with tuples from another relation S. A two-way semijoin aims to simultaneously reduce both R and S so that they each contain only tuples that match the other relation.

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 $\pi_X R_j$ to site i, (b) perform $R_i \ltimes \pi_X R_j$ yielding a new relation P, then (c) send back to site j the relation $\pi_X P$ for a restriction of R_j. We refer to steps (a) and (b) as the "forward reduction" phase, and to step (c) as the "backward reduction" phase.

In [RK91], an improved technique for two-way semijoins was proposed. We refer to this technique as "2SJ+." In the backward reduction phase, the set of matching values, or the set of nonmatching values if this set is smaller, is sent back to reduce the other relation.

In [LR94], this technique was further improved using tuple bit-vectors, as illustrated below. This new technique we call "2SJ++."

Definition 2.1: 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 ith bit of the array corresponds to the ith tuple of R. $\square$

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

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

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 $\pi_X P$ 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.

2.2 Improving Hash Filters with Tuple Bit-Vectors

Hash filters [Blo70, Bab79] can be used in a fashion similar to semijoin projections. Rather than transmitting the values for the join attribute, one hashes the values into a hash table and transmits the table. The hash table is often significantly smaller than the total size of the projection, although they do not give perfect reduction information due to hash collisions.

Similar tuple bit-vector techniques can be used in a backward reduction phase for hash filters. Imagine that the forward reduction uses a hash table N bits long that has M bits set. Instead of returning a hash table H that is N bits long in the backward reduction phase, we return a table T that is M bits long. The ith bit of T is set if the ith 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 (which is usually the case in order to minimize collisions [LR94]).

Thus we 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 A Novel DQP Client-Server Architecture

The basic architecture is illustrated below in Example 3.1. (Further refinements are presented later.) Basically we transform the N-way join into a projection join operation on the join attribute values only and a new "concatenation" operation.

Definition 3.1: Let M be the canonical representation of an N-way join. The *concatenation* operation is the mapping from M to the actual join result. □

Concatenation can be defined similarly if we start from the join result tuple connector rather than from the canonical representation.

The difference between concatenation and join is that the concatenation operation takes a canonical representation of the already computed join result (with each column representing the list of TIDs of those matched tuples in the corresponding relation). We will show later that this new algebraic operation can be performed much more cheaply than a normal join operation.

One important point to note about tuple identifiers in our case is that we can use them without transmitting them. We do not have to transmit TIDs; instead we use the tuple ordering to determine implicit TIDs. The first tuple transmitted has TID 1, the second has TID 2, and so on. As long as the transmitting site remembers the order in which the tuples were sent (which is necessary to use the tuple bit-vector techniques anyway) the client and the server can accurately refer to their tuples using the implicit TIDs.

Example 3.1: Consider a chain query $R_1(A, B, C, D, X, Y) \bowtie R_2(Y, E, F, G, Z) \bowtie R_3(Z, H, J, W)$ among three equally sized relations distributed over three remote server sites, site 1, site 2 and site 3 respectively. The client at site 4 expects the result of the form $\pi_{A,B,C,E,X,Y,W}(R_1 \bowtie R_2 \bowtie R_3)$.

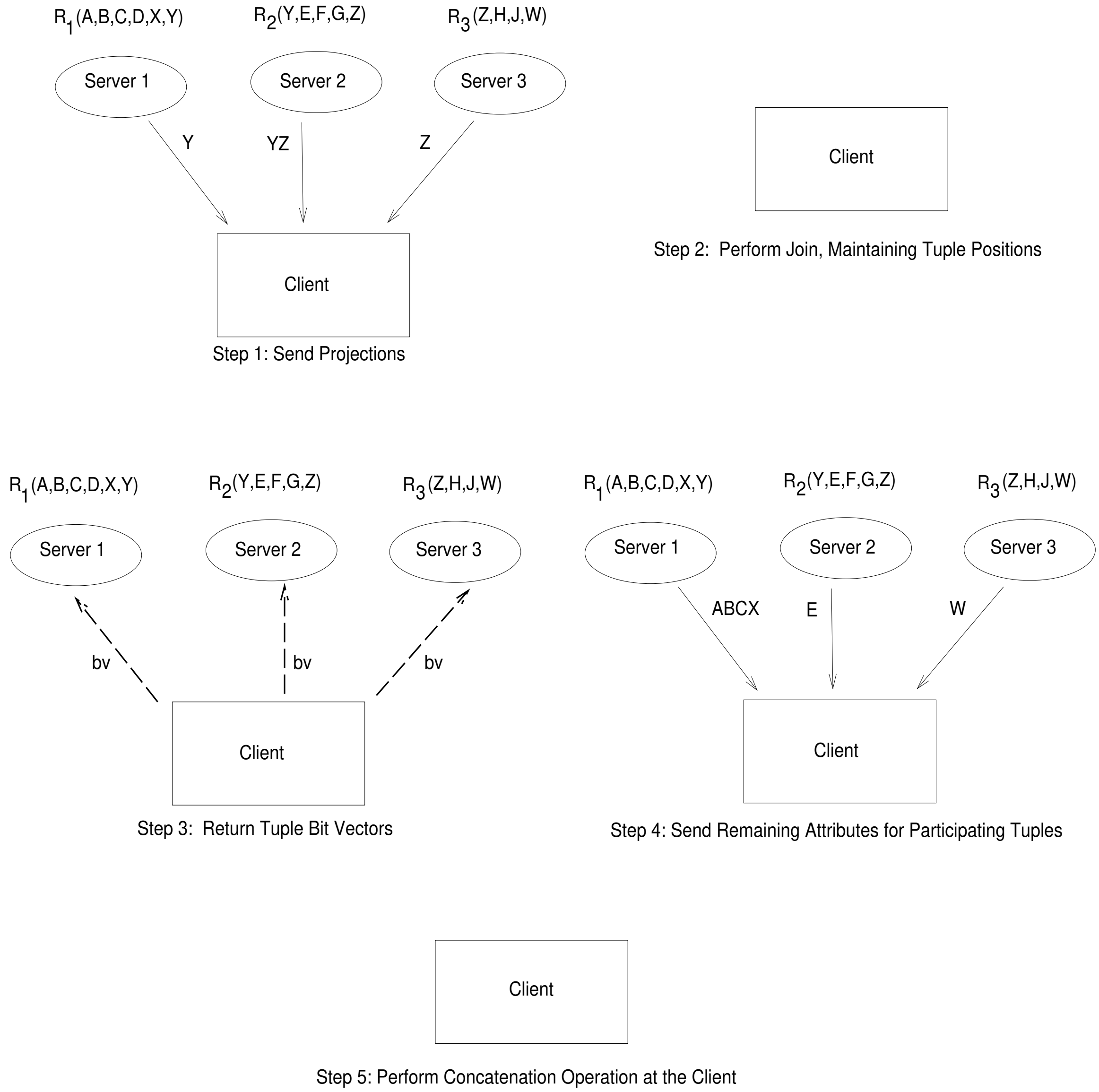

Figure 1: The Architecture

The basic steps involved in executing the query are listed below:

1. All three sites involved in the 3-way join query perform local projections and generate the projections (preserving duplicates) on their join attributes in parallel, and send them out to either a chosen assembling site or to some other processing sites in parallel. In our case, site 1 sends $\pi_Y R_1$, site 2 sends $\pi_{YZ} R_2$, site 3 sends $\pi_Z R_3$ to the client site where user submits the query.

2. The assembling site (typically the query originating site) receives all the (reduced) join projections and performs a join among them. While performing the join, the tuple bit vectors corresponding to the joining relations are built at the same time. Imagine every join projection has an implicit column which has values being the logical tuple numbers within that projection (i.e., tuple i is of the form (TID_i, i)). After the join is done, we have a join result tuple connector with values being the matched tuples' ordering indexes in each tuple bit vector, and the join attribute values that are present in the result.

3. Once the tuple connectors are constructed, we send them back to the remote sites. Note that the ordering of these tuple bit vectors are identical to those original scan ordering at each remote site. So if the original scan is an index scan, the second pass scan can still utilize the available index.

4. After each site receives its tuple bit vector, it scans the joining relation once, filters out the tuples that do not participate in the join result based on information encoded in the received tuple bit vector, and ships the matched tuples to the assembling site.

5. Once the assembling site receives all the result tuples from the joining sites, it performs a concatenation operation to construct the final join result.

Note that if we didn't want W in the result, R_3 would contribute to the join reduction but not to the final join result, so there would be no need to send back to R_3 its tuple bit vector. $\Box$

In order for the tuple bit vector technique to work, remote sites are required to send the actual join projection values and keep the necessary scan state information in order to reconstruct the original scan order. The tuple bit vector would always be a set of bits that filters out a subset of the original relation tuples. In [LR94] we show that the network overhead incurred by tuple bit vectors is very small compared with other semijoin variants.

3.1 Advantages of this Architecture

One important aspect of this architecture is that we can handle cyclic join queries. Semijoin based algorithms will not work since cyclic queries do not have full-reducer semijoin programs [BC81]. The tuple bit vectors sent back to each joining relation encode the complete join reduction information collected after constructing the full join result tuple connector. Thus our algorithm does not depend on the acyclic query property. In contrast, semijoin programs only carry a subset of the join reduction information, so it is possible for semijoin based algorithms to send spurious tuples to the assembly site without reducing the joining relations to the maximum degree, which adds extra network transmission overhead.

In a distributed heterogeneous database environment, it is typically the case that the query optimizer has only partial and possibly stale information about cost parameters at remote sites. The optimal execution plans generated by semijoin based algorithms rely heavily on the correct estimation of intermediate join result sizes, therefore are not robust to estimation errors [IC91]. Because our architecture does not depend on the semijoin technique, the importance of correctly estimating the sizes of intermediate relations resulting from a join or semijoin is significantly reduced. No error propagation in the estimation is incurred.

In addition, maximum site autonomy is maintained. Our architecture treats each remote database system as page servers [DMFV90], and is relatively insensitive to heterogeneous administration policy and data models at the remote sites. Remote DBMS are only required to provide the services of filtered relation scan for this architecture to work. So an object oriented database system can be easily incorporated in this architecture as long as it provides the file scan functionality. There is no requirement on the remote sites to always export accurate statistics as our algorithm is relatively insensitive to these statistics anyway. No inter-site connection is needed between remote sites, and thus we do not need to make assumptions about the availability of such connections. No inter-site communication traffic is incurred. Finally it imposes little processing load on the remote servers. Usually two relation scans on each server would suffice.

This architecture does not require connections between the remote servers. This can be very important for accounting purposes. The connections between remote servers are not visible to the client, who is presumably paying for the data service. With our architecture, the only connections are between the client and the remote servers. Thus the client can easily monitor the actual network traffic and verify the cost attached to answering the given query.

Even though remote DBMS servers are typically configured with large main memory cache and fast disk devices with sufficient storage capacity, they are likely to be overloaded most of the time. By only requiring them to provide the lightweight service of filtered relation scan and not dispatching time consuming join operations on them, we obtain two benefits. Firstly, since the clients do most of the work, the servers can be better utilized by more clients operating concurrently.

Secondly, the optimizer usually depends on the estimated response time of remote servers in its optimization cost model. If the remote servers are heavily loaded and manifest very unpredictable response time, the optimality of the query plan generated becomes questionable. Since remote DBMS should be full-fledged database systems capable of handling concurrency control and recovery, their query engines may not be very efficient due to these maintenance overhead (such as acquiring locks, blocking for locks etc. [Moh92]). In our architecture, the query originating site is the only site required to have query processing capabilities. And we could use specially designed high performance query engine such as parallel, main memory based system to serve this need.

3.2 Refinements

Our basic architecture needs several refinements in order to have competitive performance characteristics.

One potential problem occurs when the relations at remote sites have sizes that differ significantly. We may end up waiting too long for the larger relation's join attributes, when we could have reduced the size of the join attribute projection with a semijoin from the smaller relation.

We wish to achieve this kind of semijoin reduction without requiring a direct connection between the remote sites, for the reasons outlined in the previous section. In order to get such a reduction we can use the client as an intermediary between the remote sites. With some database statistics from the remote sites, the client will know when there is a large disparity in relation sizes, and can pipeline the values from one remote site to another for use in a semijoin, in order to achieve better overall performance.

Alternatively, one could provide a direct connection (if one was available) between two remote sites to apply a semijoin or a hash filter. In principle, this would require one network hop rather than two, but would not preserve site autonomy as discussed above.

The other potential bottlenecks are the join operation and the concatenation operation, which are done at the client site. In order to speed these operations up, we can use parallelism. We can divide the join and concatenation work among many slave processors and (with reasonable load-balancing) get a linear speedup. We discuss this issue in the next section.

4 Parallel Processing and Load Balancing

Processing distributed N-way join queries efficiently on extremely large data sets requires a huge amount of CPU and memory resources. One design objective of this architecture is to utilize the aggregate computing resources of network clustered workstations to meet this demand. In a typical configuration, there would be a master process running at the query originating site which manages a virtual pool of lightly loaded slave workstations. Each slave host can dynamically join and quit the pool, depending on its load threshold. At any moment, the aggregate computing power of the virtual pool can be fully utilized to process the query. The master and slaves are interconnected via a fast local-area network.

4.1 Parallelizing the Projection Join

Given the configuration of our architecture, there are two schemes applicable to parallelize the first round N-way projection join. One was presented in the FR algorithm [ESW78] and the Symmetric FR algorithm [SY93]. They basically unicast the whole of the largest relation and multicast the whole of the remaining $N-1$ relations to each participating slave site. Since each slave only receives a disjoint portion of the work, they can proceed in parallel without inter-slave synchronization. This scheme is very costly in network transmission because the whole relations are transmitted, thus unsuitable for our architecture which assumes a low bandwidth wide area network to remote sites.

The second scheme is to construct a dataflow pipeline using hybrid-hash join algorithms. This scheme, defined as the scheduling problem among left-deep, right-deep and bushy query tree formats used to execute the N-way join query, has been an active research topic [SD90, WA91, LCRY93, CLYY92]. We give an example to show how the first round projection join can be parallelized using this scheme.

Example 4.1: Consider the equality join

$$\pi_{AXBYCD}(R_1(A,X) \bowtie R_2(B,X,Y) \bowtie R_3(C,Y,Z) \bowtie R_4(Z,D))$$

with the tuple instances shown in Fig 2.

In the diagram shown, for the example 4-way equality join we could allocate 4 slave sites to receive $\pi_X R_1$, $\pi_{XY} R_2$, $\pi_{YZ} R_3$ and $\pi_Z R_4$ respectively and build the hash tables for R_1, R_2 and R_3 at each slave site in parallel. Each of the hash tables would have an entry of the form (Join Attribute Value, TID). Slave 4 in the dataflow pipeline also receives $\pi_Z R_4$ in parallel which acts as the probe input.[1] This way we can construct a three stage dataflow pipeline at the assembling site (or the query site) and achieve parallel speedup for the projection join operation. The join result tuple connector is also shown in the diagram. For example, the second entry $(r4, s2, t4, p1, 20, d)$ means the fourth tuple in R_1, the second tuple in R_2, the fourth tuple in R_3 and the first tuple in R_4 would form a join result tuple together with X-value 20 and Y-value d. The orders referred to correspond to the original scan orders of generating the join projections. □

One potential problem with the above scheme is that hash table overflow could occur at certain pipeline stages. They would form the bottlenecks in the pipeline throughput and degrade join performance significantly. For example, let's consider a two-stage pipeline with R at stage 1 being the probe input and S at stage 2 being the build input having one in-memory and one disk-resident overflowed hash partition (denoted as P_M, P_D respectively). Suppose the first probe tuple from R references P_D, the next probe tuple of R references P_M (which was just paged out to disk due to swapping P_D in memory), and the reference pattern $P_M, P_D, \ldots$ continues. This "thrashing" of hash partitions might involve significant disk I/O overhead which could defeat all the benefits of parallelism.

Previous research [SD90, WA91, LCRY93, CLYY92] usually focused on a multiprocessor environment. Due to the limited resources of multiprocessor machines, the probability of hash table

[1] We can either spool $\pi_Z R_4$ to slave 4's disk then initiate the probing process, or send R_4's tuple packet across the network and stored in slave 4's memory on a demand basis.

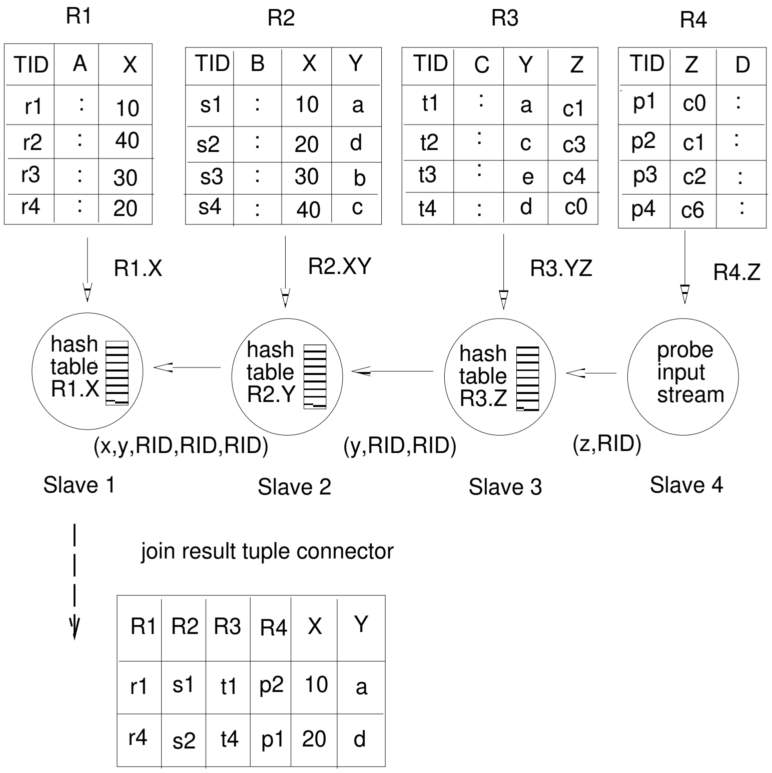

Figure 2: The Graph for Parallelizing The First Round Join

overflows given a large joining relation R is not negligible. In contrast, our architecture utilizes a dynamically configured virtual pool of slave machines having a highly scalable aggregate amount of computing resources (CPU, memory). We can always allocate a sufficient number of slaves to hold partitions of R. Thus it is almost always the case that hash tables of all joining relations be entirely resident in the aggregate distributed memory of the slave machines. If hash table overflow is unavoidable using the above strategy, the optimizer based on our architecture can allocate R to an intermediate pipeline stage, tune the partition functions so that some slaves store the partitions of R in-memory and some store R's overflowed partitions on disk. The probing tuples coming from the previous pipeline stage could be routed differently based on their reference locality, thus the queue lengths at the overflowed slave sites are effectively shortened. To the best of our knowledge, this problem has not been seriously addressed in previous work.

In summary, we can construct a dataflow pipeline to parallelize the projection join. Compared with the previous work on parallel join processing architectures, the problem of hash table overflow is alleviated significantly in our architecture. A near-zero probability of hash table overflow in our architecture also means the hash join algorithms are more likely to reach their peak performance.

4.2 Parallelizing the Final Concatenation Operation

From Definition 3.1, we know the concatenation operator takes the join result tuple connector and sets of reduced relation tuples forming the join result as inputs. Given the already computed join result information encoded in the join result tuple connector, it is expected that a concatenation shall be performed more cheaply than a join. We will show below how linear costs in terms of number of scans over each reduced relation can be achieved, and how parallelism can be applied to parallelize the concatenation operation.

The dataflow pipeline used to parallelize the projection join generates a stream of join result tuple connectors at its last pipeline stage. One strategy is to store each column of a join result tuple connector at its corresponding slave processor (which also stores the corresponding relation's join projection). After the projection join is done, each slave processor would have a complete *vertical partition* of the join result tuple connector with the same order as the join result tuple order. The value of each partition is a list of tuple position values of matched tuples in the corresponding relation together with some join attribute values. A tuple bit vector is then built using each partition and sent to its corresponding remote site. Each remote relation would be scanned once and its reduced tuples would be retrieved and sent back to each corresponding slave processor for the concatenation operation. Figure 3 illustrates this basic strategy.

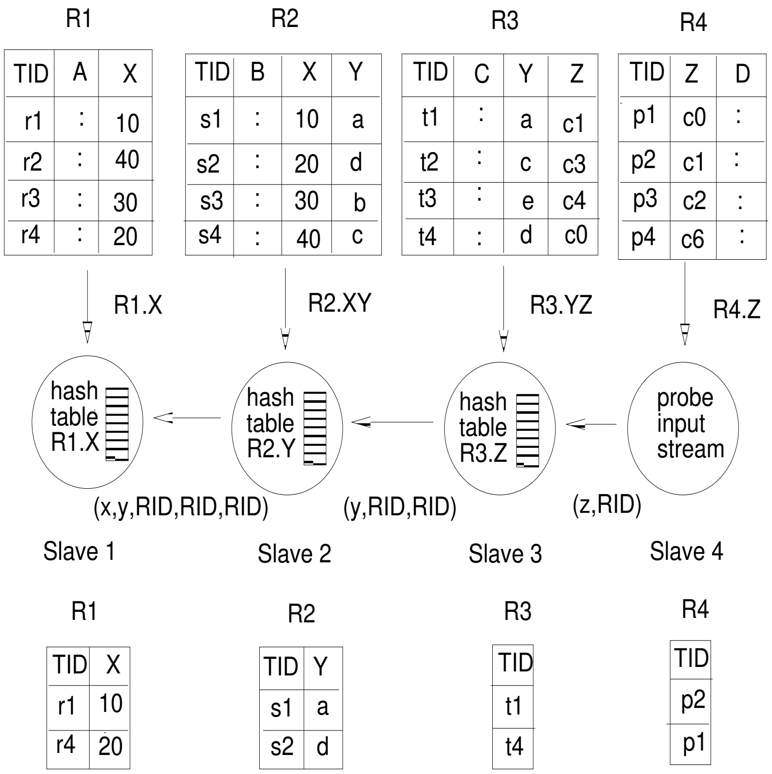

Figure 3: The Graph for Parallelizing the Concatenation

It is typically the case that there are fewer incoming tuples at each slave site than there were values of join attributes sent initially. In order to preserve the linear cost of concatenation, we build a mapping table between the tuple positions and their corresponding incoming tuples for random retrieval purposes. The mapping table can be built using the tuple bit vector alone. When the actual tuples arrive from the remote site, they are stored sequentially in an array. The mapping table can be used to index into this array given the index of the original join attribute value.

Figure 4 illustrates this technique in more detail.

In this example, slave i holds the vertical partition of the join result tuple connector $V_i = (4, 2, 6)$ corresponding to relation R_i. The tuple bit vector sent is 0101010 corresponding to the second, fourth and sixth values of the join attribute. Using the tuple bit-vector it is clear that the first tuple returned will correspond to the second join attribute value, that the second tuple returned will correspond to the fourth join attribute value, and that the third tuple returned will correspond to the sixth join attribute value. This information is represented in the mapping table.

The incoming tuples (b, e, h) are ordered differently from V_i. Thus we need to be able to randomly access tuples sent (in constant time) rather than rely on processing them in sequence. Hence the mapping table is necessary if we wish to perform concatenation in linear time.

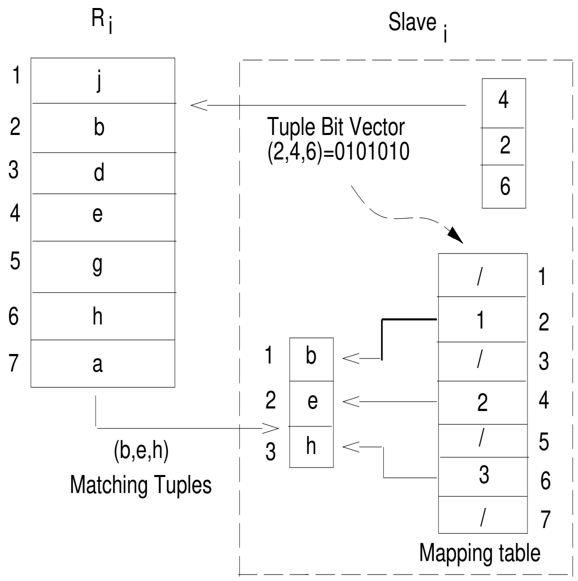

Figure 4: Constructing the Mapping Table

One potential problem with the above technique is the existence of gaps in the mapping table if the number of matched tuples is small compared with the original relation cardinality. Fortunately this is not a problem since the extra level of indirection can be processed before the actual tuples arrive from the remote sites. Thus, by the time all the tuples arrive from the remote sites we can have the join value tuple connectors referring to the local physical tuple locations, and not to the original locations that were based on the remote site ordering of the join attribute.

If the slave processor has to spill the incoming tuples to disk, then the scan utilizing the above mapping table would resemble a non-clustered index scan. In this case, a smart buffer manager is critical.

Other strategies to parallelize the concatenation operation exist. One is to partition each output N-arity join result tuple connector into N tuple positions, pipeline the transmission of these position values to their corresponding remote sites and retrieve the relevant tuples. Although having the property of faster response time for the first few join result tuples, this scheme requires multiple scanning of the remote relations and is likely to yield inferior performance. The second strategy is to horizontally partition the join result tuple connectors among a set of slave processors. The advantage is that each slave processor would have a disjoint subset of the join result tuple connector, and they can carry out the concatenation operation independently without requiring inter-slave synchronization. One disadvantage is the partitioning scheme might cause certain tuples to be replicated at some slave sites. This means the remote site has to multicast these replicated tuples to their slave sites which could incur additional network overhead. We plan to investigate the tradeoffs present in these strategies in future work.

4.3 Potential Bottlenecks for Parallel Execution

There are three major stages where load imbalances could occur, the initial transmission of join projections, the hash join probing phase and the final transmission of relevant input tuples. We addressed the first of these in Section 3.2. In this section we discuss the parallel join and parallel concatenation bottlenecks.

It the slave sites receive join projections with different sizes, uneven CPU load and memory consumption could easily happen and lead to bottlenecks in the hash join pipeline.

The dynamic nature of our master/slave paradigm to parallelize the projection join facilitates load balancing. Provided the aggregate available memory in the cluster of slave machines is enough to accommodate the hash tables for those join projections, we can fragment a big join projection and allocate multiple slave processors to hold the hash tables of its fragments. We can use round-robin, range or hash partition strategies to achieve this. If tuples in the join projection is distributed in a round-robin fashion, the join load at each slave machine is easily balanced. However, each output tuple connector generated by the previous stage of the dataflow pipeline has to be multicasted to the set of slave processors holding the fragment's hash tables, this might incur extra join and network load. If range or hash partitioning is used, each output probing tuple connector has to be hashed or range partitioned on the partitioning attribute and unicasted to the appropriate slave processor. In this case, tuple bit vector technique still works, although this requires involved maintenance of the relative tuple ordering at each slave site and the partitioning information at each remote site.

If memory is scarce, the optimizer could allocate one slave processor to hold several join projections provided they together can fit into this slave processor's memory. Otherwise the hash tables have to be spooled to disk.

Due to the quadratic nature of the join operation, the intermediate result sizes corresponding to one probing input can grow polynomially in the worst case. This is basically an optimal join ordering problem. By ordering the relations at different stages of the dataflow pipeline, we can avoid cartesian products and minimizing the intermediate result sizes. The probing relation should be the one with maximum reduction effect. It would be ideal if the join selectivities of each relation are in ascending order from the beginning of the dataflow pipeline to the end. This way the total number of intermediate tuple connectors shipped across the pipeline is minimized.

The backward reduction phase which consists only of the tuple bit vectors can be estimated accurately by the optimizer because their sizes are available very cheaply at run-time. Also we only depend on the scanning time parameter of those remote servers, which are relatively stable and robust parameters (e.g., once the disk is attached to the remote server, scanning time is relatively fixed).

5 Performance Improvements

In a sense, our architecture is at a potential disadvantage compared with architectures that (unrealistically) assume a connection between the remote sites, since there is an additional data path in these alternative proposals. Nevertheless, there are situations in which our architecture outperforms some of these proposals.

This architecture can be a winner compared with the traditional semijoin based three-stage architecture to process N-way join, for instance, compared with the "One-Shot" algorithm presented in [WCS92]. In our architecture, we construct the full join in the tuple connector form. We then send back the tuple bit vectors, which effectively eliminates *all* the unmatched tuples at each remote site. In contrast, semijoin based approaches may eliminate only a *subset* of the unmatched tuples.

Example 5.1: Suppose we are dealing with $R(A,X) \bowtie S(B,X,Y) \bowtie T(Y,C)$ where A, B, C are the set of result attributes. Assume X, Y are key attributes in all three relations (thus duplicates play no role in this example). Let $W(X) = W(Y) = 4\ bytes$, $W(A) = W(C) = 96\ bytes$, and $W(B) = 92$ bytes, so that tuples in R, S and T are 100 bytes long, and let $|R| = |S| = |T| = K$. $\pi_X R$ has values of the form n, $\pi_{XY} S$ has values of the form $(2n, 2n)$, and $\pi_Y T$ has values of the form $n + K$, for $1 \leq n \leq K$. The following table illustrates the instances of these relations given $K = 100$.

R

A	X
	1
	2
	3
	⋮
	50
	51
	⋮
	99
	100

S

B	X	Y
	2	2
	4	4
	6	6
	⋮	⋮
	100	100
	102	102
	⋮	⋮
	198	198
	200	200

T

Y	C
101	
102	
103	
⋮	
150	
151	
⋮	
199	
200	

R and T would be reduced in size by half after semijoining with S separately, and S would be empty after semijoining together with both R and T. The basic flaw of the one-shot algorithm is that for examples like this, one shot is not enough. Despite the semijoin, the reduction effect is far from optimal. We now analyze the cost (i.e., response time, counting just the network transmission time in bytes) of the one-shot algorithm on this example.

As described in [WCS92], the one-shot algorithm uses hash filters to compress the semijoin information in order to reduce the network overhead. In [LR94], we argue that the number of bits used in the hash filters, i.e., the size of the hash table, should be at least equal to the cardinality of the larger joining relation, in order to reduce the effect of hash collisions. So, suppose that $K = 10^6$, and let the hash filters be 10^6 bits long. The optimal strategy produced by the one-shot algorithm would perform these hash filter based semijoins in parallel: $R \ltimes S$, $S \ltimes R$, $S \ltimes T$, $T \ltimes S$. The total response time includes the transmission time of hash filters and the transmission time for reduced R and T, and is given by:

$$\frac{10^6}{8} + \frac{10^6}{2} * 100 \approx 5 * 10^7 \text{ bytes.}$$

Our architecture would send all the join projections to the assembling site, and send back their tuple bit vectors if the projection join is not empty. In this example, the projection join is empty, so the total response time is $8 * 10^6$, a factor of 6 better.

Note that by translating the tuples in $\pi_X R$, we can slowly increase the number of matched tuples in S up to $10^6/2$. In this case, the response time of one-shot would stay constant, and our architecture would send back $10^6/8$ bits of tuple bit vectors and receive the reduced R, S, T tuples. Let ρ_S be the join selectivity of $S \bowtie T$ on R (or $S \bowtie R$ on T). By solving the following inequality

$$8 * 10^6 + \frac{10^6}{8} + \rho_S * 100 * 10^6 \quad \leq \quad \frac{10^6}{8} + \frac{10^6}{2} * 100$$

we can conclude that when $\rho_S \leq 0.42$ our architecture would win.

The reason for one-shot's poor response time is that in order to minimize response time, one-shot considers only parallel execution of semijoin programs without any sequential propagations. This means the reduction effect from T (or R) to S is not propagated to R (or T); thus one-shot would always send half of R and T, which contain spurious tuples, to the assembling site. In contrast, using our architecture, we pay the overhead of sending the join projections and their tuple bit vectors, but receive optimal number of relation tuples back at the assembling site because we utilize the whole join reduction information. □

The class of cyclic join queries doesn't have a full-reducer [BC81]. For this class of queries, semijoin based algorithms would fail to perform effective network cost reduction and little optimization could be done to reduce the total query processing and response time. In contrast, our architecture computes a full join of the projected relations, and is capable of fully reducing cyclic join queries as well.

Example 5.2: Suppose we are dealing with $R(A, X, Y) \bowtie S(B, Y, Z) \bowtie T(C, Z, X)$ where A, B, C are the set of result attributes.[2] Suppose the current join attribute values of these relations are $XY = \{a_1b_1, a_2b_2, \ldots, a_nb_n\}$, $YZ = \{b_1c_1, b_2c_2, \ldots, b_nc_n\}$, $ZX = \{a_2c_1, a_3c_2, \ldots, a_nc_{n-1}, a_1c_n\}$, for some n. A moment's reflection tells us that the join of these three relations is empty, and no semijoin step can change any of the three relations. Thus, not only is there no one semijoin program that reduces the relations independent of their initial values, but there is no semijoin program at all that works for this initial database.

Assume each of the join attributes X, Y and Z takes four bytes and A, B, C each takes 92 bytes. Let n be 10^4. The one-shot algorithm would send all of R, S, and T in parallel to the assembling site. The response time (even ignoring the hash-filter cost) is

$$100 * 10^4 = 10^6$$

Our algorithm would send XY, YZ, ZX first and realize there are no input tuples to form the results. The response time is

$$8 * 10^4.$$

We achieve a factor of twelve saving in terms of response time.

In our architecture which is not based on semijoin operators, we construct the full join result tuple connector first by sending XY, YZ, ZX to the assembling site. Knowing the final join connector is empty, we don't need to send back the tuple bit vectors at all. The overhead we pay in this example is only the network cost of join projections instead of the whole tuple values. □

6 Survey of Related Work

In [RK91] the concept of tuple connector is used to construct a pipeline to handle an N-way join query. An improved semijoin variant denoted as 2SJ+ [LR94] is combined with caching to obtain a pipelined N-way join algorithm. Because all nonjoining attributes are not included in the tuple connectors during the semijoin (or join) reduction stage, they are typically a lot smaller than their corresponding relations, thus incurring less network overhead. Because of their compact size, tuple connectors can usually be stored in unnormalized main memory data structures that further reduce their size and access time. The main goal of the algorithm is to eliminate the need of shipping, storing, and retrieving foreign relations and/or intermediate results on the local disks of the remote and the query site during the processing of the N-way join.

Our work differs from [RK91] in that we use an even better primitive based on the idea of the tuple bit-vector [LR94]. Instead of sending back the matched (or unmatched if the number is smaller) join attribute values during the backward reduction phase, we send a much more compact bit vector with bits corresponding to matched (or unmatched) tuples set to 1 to deliver the same reduction information. The tuple bit vector is positional in nature so it is possible to eliminate the need to ship the actual TID values because we can overload the tuple ordering (record numbers) within the join projections as logical TIDs and encode this ordering information for those matched tuples for future tuple retrievals. Thus we pay much less network overhead during the forward transmission phase of join projections. After constructing the join result tuple connector, [RK91] sends back the actual TIDs for the matched tuples. In contrast, we send back a tuple bit vector

[2] This example is based on one from [Ull89].

which is usually cheaper in network cost.[3] Also our pipeline model to construct the join result tuple connector is more sophisticated and employs parallelism aggressively. Parallelism issue is not addressed in [RK91].

In [SD90, WA91, LCRY93, CLYY92], the problem of pipelined hash-based processing of N-way join queries in a tightly coupled, shared-nothing multiprocessor database environment is investigated. It is demonstrated that right-deep scheduling strategies can generally provide significant performance advantages in a large multiprocessor database machine compared with left-deep scheduling strategies, even when the aggregate memory is limited.

The environment we are investigating is a distributed, heterogeneous environment with participating sites interconnected through a high latency, low bandwidth wide area network. So reducing the network cost when reducing and moving relations to the assembling site before starting the actual join operation is the major goal. In a multiprocessor environment the interconnection network typically has a very high bandwidth. Data distribution is relatively cheap, and efficiently utilizing the memory is usually the chief optimization criterion. For instance in [LCRY93], only hash table building and join probing costs are considered which primarily consist of CPU costs only. In our scheme the calculation of hash table building costs would have to explicitly take into account the relation scanning time and network transmission cost. We have shown in previous sections how we tailored these pipelined hash-join techniques to parallelize the first stage projection join, and to seamlessly integrate the pipeline with the second stage concatenation operation.

7 Conclusions

The main contribution of this paper is the proposal of a novel client-server architecture to process distributed N-way join queries in a distributed, heterogeneous database environment. This architecture is highly scalable by exploiting the aggregate computing resources of a cluster of networked machines to parallelize the expensive join processing. It also has the advantages of maintaining maximal remote site autonomy and robust to inaccurate database statistics. By adopting the tuple bit vector idea rather than the commonly used semijoin technique, the network overhead is more effectively reduced.

We compared our tuple bit vector based DQP algorithm within this architecture with other semijoin based DQP algorithms. Using the “one-shot” algorithm [WCS92] as a representative of parallel semijoin based DQP algorithms, we demonstrated that our algorithm can lead to better performance for both commonly encountered chain queries and cyclic queries than conventional DQP algorithms. Note the ideas presented in this paper are, in principle, also applicable to other distributed query processing algorithms, and could be easily implemented within them to enhance their performance.

In future work, we plan to implement this architecture in the context of a distributed relational query processing system being developed at Columbia University. We hope to demonstrate the *practical* utility of this architecture, 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.

[3] Even if we send the tuple ordering indexes of the set bits in the tuple bit vector as a means of compression, the size of the ordering indexes is bounded by log(Join Projection Size). So a 4 byte address can represent 4GB join projection. On the other hand, physical TIDs are usually constrained to be a multiple of the machine word size.

[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.

[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.

[CLYY92] M.-S. Chen, M.-L. Lo, P.S. Yu, and Y.C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of the 18th International Conference on Very Large Data bases*, pages 15–26, 1992.

[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.

[DMFV90] David J. DeWitt, David Maier, Philippe Futtersack, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th VLDB conference*, pages 107–121, 1990.

[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.

[IC91] Y. Ioannnidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 268–277, 1991.

[LCRY93] Ming-Ling Lo, Ming-Syan Chen, C. V. Ravishankar, and Philip S. Yu. On optimal processor allocation to support pipelined hash joins. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 69–78, 1993.

[LR94] Zhe Li and Ken Ross. Better semijoins using tuple bit-vectors. Technical Report CUCS-010-94, Columbia University, New York, NY 10027, 1994.

[Moh92] C. Mohan. Interactions between query optimization and concurrency control. In *Second International Workshop on Research Issues on Data Engineering: Transaction Processing and Query Processing*, pages 26–35, 1992.

[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.

[SALP79] P. Griffiths Selinger, M. M. Astrahan, R.A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 SIGMOD Conference*, pages 23–34, 1979.

[SD90] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th VLDB conference*, pages 469–480, 1990.

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

[SY93] James W. Stamos and Honesty C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Transactions on Parallel And Distributed Systems*, 4(12):1345–1354, 1993.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).

[WA91] A. Wilschut and P. Apers. Dataflow query execution in parallel main-memory environment. In *Proceedings of 1st conference on parallel and distributed information systems*, pages 68–97, 1991.

[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.

[Yao77] S.B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.

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