Query Processing in a Symmetric Parallel Environment

by

Dennis Shasha

Ultracomputer Note #95
Technical Report #197
January, 1986

This work was supported in part by National Science Foundation Grant DCR-85-01611 and by Office of Naval Research Grant N00014-85-K-0046.

ABSTRACT

We consider a database machine consisting of n nodes connected by an O(n*processing speed) bandwidth network. Each node consists of a processor, a random access memory, and a slower but much larger memory such as a disk. In order to approach optimal (O(n)) speedup on this hardware architecture, we partition relations roughly evenly among the processors. We study the problem of optimizing multi-join queries assuming such a data distribution strategy. For us, optimization consists of minimizing the number of times we need to redistribute relations in the course of a query. The optimization problem is NP-complete for general queries but linear time for a subclass of tree queries.

1. Setting and Problem

The goal of the New York University Ultrabase project is to discover architectures and algorithms that speed up relational query processing roughly linearly with the number of processors.[1] Fortunately for our choice of project title, we choose a symmetric parallel architecture (figure 1), generalizing the architecture of the New York University Ultracomputer [GGKARS83]. In this architecture, each processing node consists of a processor, a random access memory, and a slower but much larger memory. They are connected by a network whose bandwidth is proportional to the number of processors (That is, the network can sustain a communication load in which each processing node sends a message to an arbitrary processing node every c cycles, on the average, for some c unrelated to n.)

We justify our choice by an intuitive symmetry argument. To achieve an optimal speedup of, say, a join of R and S with the join clause R.A = S.B, each of the n processing nodes of our database machine should have about the same amount of work to do. This suggests that each processing node should contain approximately 1/n of the R and S tuples and should perform the join on those. In order for the result of the join to be correct, no tuple of R in processing node i should have the same A value as the B value of some S tuple in another processing node j.[2] This implies that it may sometimes be necessary to send tuples from one processing node to another. For example, suppose the R tuples are *partitioned* on A (i.e. all R tuples with the same A value are in the same processing node), but the S tuples are not partitioned on B. It may be necessary to send nearly every tuple of the S relation from one processing node to another in the network so the S tuples are partitioned on B and each S tuple is in the same processor as the R tuples it will join with. This step is called *repartitioning.* If all the S tuples are initially in one processing node, then the repartitioning will take time proportional to the time it takes for that node to send its tuples. This would not give us optimal speedup. Therefore, we assume that the S tuples are initially partitioned as well (though not necessarily on B). Since it may be necessary to send nearly every tuple of the

[1] A secondary goal is to achieve this speedup without sacrificing fast execution of simple transactions or concurrency control. We will deal with these aspects in subsequent papers.

[2] If 1/2 of the R tuples have the same A value, then we will certainly not attain optimal speedup. Such cases are probably rare, since most joins involve keys and distributions are seldom so perverse even on non-key fields.

S relation, the bandwidth of the network should be O(n*processing speed).

We conclude that we need a high bandwidth network and we need to partition the relations.[3] The NYU Ultracomputer, the IBM Research Prototype 3 [PBGHKMMNW85], and NONVON [HSN] have these architectural properties. Other database machines such as [DG85, KTM84, KTM83, T84, VG84] can be roughly modeled this way if one assumes that the networks they use have sufficient bandwidth for the number of processing nodes they have.[4] Our work therefore builds on the join algorithms already proposed for those machines. (See [BR85, H83] for a modern review of database machines and original papers [B79, BDFW83, BO79, D79, GS81, MH81, S79, Schm79, SG75].)

For the purposes of this paper any data distribution strategy that partitions a relation roughly evenly based on some subset of its attributes will be satisfactory. The one we use is hash partitioning. To partition a relation R based on attribute A, we use a function h whose domain is the domain of R.A and whose range is the set of processing node identifiers. For each tuple t of R we put t in processor $h(t.A)$. Thus, when doing a join based on the clause R.A = S.B, if R is already partitioned based on A using hash function h, we send each S tuple x to $h(x.B)$. The idea of using hash partitioning for joins comes from [B79, KTM83, B84, DG85] and has been analyzed favorably [DG85]. In a previous report [SS85] analyzes join processing using hash partitioning, concentrating on the case when the join fields do not constitute a key. That paper explores various data reduction strategies such as having the network filter out duplicate tuples. We will return to some of the points raised in that paper in the penultimate section of this one.

1.1. Repartitioning as Cost

For the purposes of this paper, the optimum execution of a query is one which minimizes the number of repartitionings. For example, consider the query of figure 2 with join clauses R.D = S.B and S.A = T.C. Suppose that each relation is partitioned on its A attribute. Joining S with T first requires repartitioning T, but not S. Then joining R and ST requires repartitioning both relations, giving three in all. Joining R with S first requires repartitioning both R and S. Joining T with RS then requires an additional two repartitionings, giving four in all. We would therefore consider the first strategy to be better.

As justification for considering repartitioning to be the cost to be optimized, note that any reasonable strategy will require $m - 1$ joins (plus perhaps some number of selections and projections) if m relations are involved. Note also that repartitioning is expensive. Repartitioning a relation Q requires the following action at each processing node i: read the part of Q stored in the secondary memory of processing

[3] We haven't argued that other operations such as selection and projection don't suffer from this data distribution strategy. The argument is obvious for selection. Projection may require a repartitioning, but assuming a high bandwidth network, this can still be done with optimal speedup. Union, intersection, and grouping can be similarly handled. Output of an entire relation does not enjoy optimal speedup in this architecture. We can only say that the output of most queries is relatively small, so we ignore this problem.

[4] One difference is that the ring networks permit filtering strategies such as bit vector filtering [B79,DG85], because each point-to-point transmission in a ring network is in fact a broadcast.

node i and send most of the read tuples to other processors across the network. At any given receiving processing node j, it may be necessary to store some of the tuples of Q on secondary storage if the size of the current instance of Q divided by n is greater than the random access memory size of a processing node.[5] Thus, repartitioning may be more expensive than a subsequent parallel join if communication costs and, for large relations, Input/Output costs are high compared with processing costs.

The reader may complain that the repartitioning cost measure ignores obviously important parameters such as the size of a relation to be repartitioned. It would seem possible for one strategy to require fewer but vastly more expensive repartitionings than another strategy. This is possible, but our experience with a simulator has convinced us otherwise. To a first approximation, optimizations based on selectivity estimates and relation sizes can be done after the minimization of partitioning cost with little penalty. For example, once one arrives at the set of joins that minimize the number of repartitionings, one can order these joins based on other criteria, such as size. For a discussion of other such criteria, see [JK84, U82] for reviews and original papers [AHY83, BGWRR81, ES80, GS81, GS82, HY79, Schk82, WY76].

2. Terminology

A *relation scheme* R is a finite set of attributes $\{A_1, \cdots, A_n\}$. Associated with each attribute A_i is a domain denoted $dom(A_i)$. A *relation instance* r on scheme R is a finite set of mappings $\{t_1, \cdots, t_m\}$ from R to the set of domains such that for each $t \in r$, $t(A_i) \in dom(A_i)$. Let B be an attribute in R. We will write t.B for t(B). We define r.B to be $\{t_1.B, t_2.B, \cdots, t_m.B\}$.

An equi-join clause or *clause* for short is a pair of the form {R.B, S.C}, where B and C are attributes of R and S respectively. This clause represents the join condition R.B = S.C.[6] We interpret clauses of the form {R.B, R.C} to mean select only those tuples of R whose B and C values are the same. We call such a condition an *intra-relation restriction.* Performing this restriction can be done a tuple at a time and will never require a repartitioning of the relation. Intra-relation restrictions may arise in other ways as well. For example, they arise in simple selections on values (e.g. all employees in the toy department). Intra-relation restrictions also arise on the intermediate relations of a query.

---

[5] It might seem possible to avoid this by sending the tuples of Q only as they are needed in the subsequent operation. For example, one might send all tuples whose join values are between 1 and 1000 now, 1001 and 2000 in the next stage, and so on. The trouble is that since Q would not normally be sorted on an attribute it is not partitioned on, processing each stage may require a complete sweep through the Q tuples at the source processing node.

[6] We don't consider non-equi-join clauses in this paper. To process queries with non-equi-join clauses, solve the subquery with equi-join clauses first, then apply the non-equi-join clauses as restrictions on the tuples of the result relations. Since these restrictions require no repartitioning, we consider them to be inexpensive operations. For example, to find all employees who earn more than their managers, perform the equi-join Emp1.manager = Emp2.name, then apply the intra-relation restriction Emp1.sal > Emp2.sal to each tuple of the constructed relation. Note that Emp1 and Emp2 are considered to be different relations in this query.

To represent a join that links different tuples of the same relation, we consider the two arguments of the join to be different relations for the purposes of the query. For example, to find the salary of each employee's current manager, given a Emp(Name, Sal, Manager) schema, we need two instances Emp1 and Emp2 of the employee relation. These would be represented as distinct relations in the set of clauses.

For our purpose, a *query* is a set of clauses, sometimes called a *qualification.*[7]

The *query graph* for some query q is denoted QG(q). QG(q) is a pair (G, q) with the following properties. G = (V,E) where the vertices V are the relation schemes referenced by the clauses in q. E = {{R,S} | some member of q references both R and S}. Notice that the query graph is never a multigraph, though there may be many clauses associated with each edge. We write clauses({R,S}, q) to denote the set of equi-join clauses referencing both R and S in q.[8]

Notice that our definition of query graph permits the existence of *self-loops,* edges from nodes to themselves e.g. an edge from R to R resulting from the clause {R.x, R.y}. These correspond to intra-relation restrictions.

A *singleton query graph* is a query graph for a query q such that clauses(e,q) is a singleton set for each edge e in the graph. Figure 3 shows a singleton query graph.

A *spanning tree* of a query graph ((V,E), q) is a query graph ((V,E'), Treeq) with the following properties:

(1) $E' \subseteq E$, and (V,E') is a tree.

(2) For each $e \in E'$, clauses(e,Treeq) $\subseteq$ clauses(e,q) and clauses(e,Treeq) $\neq \emptyset$.

If clauses(e,treeq) is a singleton edge for each $e \in E'$, then we call ((V,E'),Treeq) a *singleton spanning tree.*

We can always execute a query by processing a singleton spanning tree of the query, then processing the remaining clauses (i.e. the ones not in the singleton spanning tree) as intra-relation restrictions. For example, we can execute the query of figure 3 by joining R and T based on R.D = T.B and RT with S based on S.C = RT.B, then the join condition S.F = R.E becomes an intra-relation restriction on the result of the second join.[9]

---

[7] Some authors include a target list in the definition of a query, where the target list denotes the attributes to appear in the result relation. In this paper, our cost measures depend on the qualification, so we ignore the target list. There are of course costs associated with the size of the target list. For example, if some attribute R.B is not in the target list and is not in any join clause, it may be projected out early. The problem of when to project attributes out in the course of a query has been well studied [AHY83, CH80, CH82].

[8] We assume throughout this paper that the query graph of a query is connected. Otherwise, we consider the queries to correspond to the connected components of the query graph; the resulting query would then be a cartesian product of the individual queries.

[9] - To disambiguate the attribute we are talking about, we keep the relation name in the clause we add to the set of intra-relation restrictions. For example, S.F = R.E applies to the F attribute contributed by the S relation and the E attribute contributed by the R relation.

The partition attributes of a relation R, *part(R)* is a subset of scheme R such that tuples of r with the same value of part(R) are in the same processing node. (To implement this as hash partitioning, we choose a hash function h and map each tuple $x \in r$ to node h(x[part(R)]).) For simplicity of presentation, we will assume that relations are partitioned on single attributes.

Our general approach is to manipulate a query and its query graph, until an execution of the clauses of its query graph yields the fewest repartitionings possible. We therefore define the repartitioning cost or simply *cost* of a query graph QG(q) as follows.

$\text{Cost}((V,E),q) = \sum_{R \varepsilon V} \text{nodecost}_{E,q}(R)$. The node cost of a relation is the number of non-partitioning attributes in join clauses touching that relation.

$\text{Nodecost}_{E,q}(R) = ||\{R.x \mid \exists\ S, \exists\ y$ such that $R \neq S$, $\{R.x, S.y\} \in$ clauses($\{R,S\}$,q) and $(\text{part}(R) \neq \{x\})\}||$.[10] Intuitively, the repartitioning cost is an upper bound on the number of repartitionings that a straightforward execution of query q would require.[11] For singleton spanning trees, it is also a lower bound.

For example, the repartitioning cost of the graph in figure 2 is one for R, S, and T, giving a total of three. The cost of the graph of figure 3 is two for R, two for S, and one for T, giving a total of five. The cost of the graph of figure 4 is one for each of R, S, and T, giving a total of three.

A *minimal repartitioning cost spanning tree* of a query graph QG(q) = ((V,E), q) is a singleton spanning tree of QG(q) such that no other singleton spanning tree of QG(q) has a smaller partition cost.

For example, figure 4 is a minimal repartitioning cost spanning tree of the query graph in figure 3. Note that the singleton spanning tree produced by removing any other edge from the query graph in figure 3 would not be a minimal repartitioning cost spanning tree, since it would have a cost of four. The clauses removed when forming a singleton spanning tree must be added to the set of intra-relation restrictions.

2.1. Join graphs, Tree Queries, and Closures

A second kind of graph associated with a query q is the *join graph* of q [BC81], denoted JG(q). The nodes of JG(q) are $\{R.x \mid R.x$ is in some clause of q$\}$ and whose edges are q, the clauses of the query. (Unlike query graphs, edges represent equality conditions in join graphs.) If R.x and S.y are in the same connected component in this graph, R.x = S.y is implied by the query. Finding connected components can be done in time proportional to the number of clauses.

Let C be a set of clauses. Let c = {R.x, S.y} for some R, S, x, and y be a clause in C. Clause $c \in C$ is *redundant* with respect to C - {c} if there exist members of C - {c} $\{R.x, T_1.z_1\}$, $\{T_1.z_1, T_2.z_2\}$,

[10] - In the general case, where part(R) and x may be a set, the last condition becomes $\neg$ (part(R) $\subseteq$ x).

[11] Straightforward, in this context, means executing q itself, not some query q' that is equivalent to q.

$\{T_2.z_2, T_3.z_3\}$, ... $\{T_m.z_m, S.y\}$. The intuition is that each of these clauses correspond to an equality condition, so c is implied by transitivity from the clauses listed. Each of these clauses corresponds to an edge in a join graph, giving us the following proposition.

Proposition: Clause c is redundant with respect to C - {c} iff c is in a cycle in JG(C). □

Two queries q and q' are *equivalent* if one may be obtained from the other by adding or removing redundant clauses.

Fact: If q and q' are equivalent, they return the same value for all database states.

The *closure* of a query q is an equivalent query, denoted q^+ to which one cannot add more redundant clauses.

Suppose a query q has a query graph QG = ((V,E), q). We say that q is tree-equivalent or is a *tree query* if it is equivalent to some query q' such that QG(q') = ((V,E'), q') and (V,E') is a tree. This definition is taken from [AHY83, GoodShmu82, BC81, CH80, CH82], where the importance of tree queries for query processing in distributed databases is discussed.

Fact: There is an $O(||q||)$ test to determine whether a query q with graph ((V,E), q) is a tree query and to construct an equivalent spanning tree if so [BC81, G79, YO79].

2.2. Eliminating Self-loops

For technical simplicity, we wish to transform given queries to equivalent queries having no self-loops in the closure. Our basic strategy is to identify implied self-loops (i.e. ones that would appear in the closure), to add those to our set of restrictions, and then to simplify the query.

(1) Find all self-loops in the closure of q. This can be done by finding all pairs R.x and R.y that are in the same connected component of the join graph of the given query.

(2) If we discover a self-loop {R.x, R.y}, replace either R.y by R.x or R.x by R.y in all the clauses of the query. The choice is arbitrary unless one of them, say R.y, is the partition attribute of R, in which case replace R.x by R.y. Add {R.x, R.y} to the set of intra-relation restrictions. Remove {R.x, R.y} from the set of clauses in the query.

For example, suppose we have clauses {R.x, S.y}, {S.y, T.z}, and {T.z, R.q}. Then {R.x, R.q} is implied, i.e. in the result, every tuple must have the same x and q value. So, we replace R.q by R.x in all the clauses where R.q appears, giving {R.q, S.y}, {S.y, T.z}, and {T.z, R.q}. We also add {R.x, R.q} to the set of intra-relation restrictions.

3. Main Results

Our goal is to minimize the number of joins and repartitionings required to process a query. As noted above, any reasonable strategy will require only m – 1 joins for an m relation query (as does ours), so we concentrate our attention on minimizing the number of repartitionings.

The terminology of the last section suggests that finding a minimal repartitioning cost spanning tree for a given query should be helpful. This is true, but one must be careful. For example, in figure 5 there are two equivalent queries T1 and T2. As they stand they have the same cost, yet a minimal repartitioning cost spanning tree of T2 has a lower cost than that of T1. This suggests the following algorithm schema:

```
proc optimize(q: query)
        begin
         map q to a query graph QG(q);
         eliminate self-loops;
         find the closure QG(q+);
         find a minimal repartitioning cost spanning
                tree of the closure;
         process the query based on that tree and
                on other heuristics
        end
```

The procedure suggests a few questions. First, is it always necessary to find the closure? (Figure 5 shows that it is necessary even for some tree queries.) Second, what is the complexity of solving the minimal repartitioning problem, i.e. of finding a minimal repartitioning cost spanning tree of a graph? Third, given a singleton spanning tree of cost k, are we always sure that we can process the query with at most k repartitionings?

Our main results are:

(1) It is NP-complete to solve the minimal repartitioning problem for general queries, even for general tree queries and the closures of those queries.

(2) For a subclass of tree queries known as single-clause tree queries, the problem is linear in q and we do not need to compute the closure.

(3) It is always possible to process a singleton spanning tree of cost k using k repartitionings. The order of joining is somewhat flexible, giving an opportunity for applying other heuristics.

3.1. NP-completeness for general queries

To establish the NP-completeness of the minimal repartitioning problem on a general query, we frame it as the following decision problem.

Instance: Query q with query graph QG(q), integer K.

Question: Is there a spanning tree of the graph of cost ≤ K?

This problem is in NP, since we can calculate the repartitioning cost of a query in polynomial time. So we can ‘‘guess’’ a spanning tree and test whether its cost is less than or equal to K in polynomial time.

To show the NP-hardness of the minimal repartitioning problem on a general query, we consider a restricted form of queries: every clause is of the form {R.x, S.x} (same attribute from both relations).

Theorem 3: The minimal repartitioning problem is NP-hard for a general query q as well as for its closure q^+.

Proof: Our reduction is from the hitting set problem [GJ79, p. 222].

Instance: Collection C of subsets of a finite set S, positive integer K.

Question: Is there a subset S' $\subseteq$ S with $||S'|| \leq K$ such that S' contains at least one element from each subset in C?

We now construct the corresponding query q with graph QG(q). We let the set S correspond to a set of attributes and members of C to correspond to relations. Thus, let there be a distinguished relation R, and relation Qi corresponding to each collection Ci in C. If Ci contains members $B_{i1}, B_{i2}, \cdots, B_{ik_i}$, then Clauses({R, Qi}, q) = {{ $R.B_{i1}, Qi.B_{i1}$} { $R.B_{i2}, Qi.B_{i2}$} { $R.B_{ik_i}, Qi.B_{ik_i}$}}. The B_{ij}'s are all distinct from partitioning attributes. See figure 6.

It follows easily from this construction that there is a minimal repartitioning cost spanning tree of QG(q) with cost at most $||C|| + K$ if and only if there is a hitting set of C with at most K elements. This completes the proof of the first part of the theorem: the minimal repartitioning problem is NP-hard for a general query q.

We now show that this condition holds for any query graph equivalent to QG(q) (e.g. $QG(q^+)$). Thus we end this proof with the following claim.

Claim: If QG(q') is equivalent to QG(q) and there is a minimal repartitioning cost spanning tree of QG(q') of cost $||C|| + K$, then there is one for QG(q) that costs no more.

Proof of claim: Let a minimal repartitioning cost spanning tree of QG(q') be called QG(Treeq'). Suppose Treeq' uses m different attributes. Note that by the construction of q, every clause in Treeq' must be of the form {Q.x, S.x} for some x, since these are the only kinds of clauses that are in q or redundant to q. We will show that the cost of Treeq' must be at least $||C|| + K$, i.e. $m \leq K$. Then we will show how to construct a singleton spanning tree of QG(q), called QG(Treeq), which is of cost $||C|| + m$, proving the claim.

Subclaim: Any non-null subset T' of Treeq' such that QG(T') is connected must have the property that cost(QG(T')) $\geq ||T'||$ + number of attributes in the clauses of T'. Hence cost(QG(Treeq')) $\geq ||C||$ + m, where m is the number of different attributes of Treeq'.

Proof of subclaim:
Base case: $||T'|| = 1$. There is only one attribute in the clause by construction of clauses. There are two nodes, so cost is 2.

Inductive step: If
(1) $||T'|| = i$ and the number of attributes in T' is j implies that cost(QG(T')) $\geq i + j$;

(2) c is of the form {Q.x,S.x} and c ∈ Treeq' – T';
(3) Q is in T'; and
(4) T' is connected;
then the claim still holds for T' ∪ {c} and QG(T' ∪ {c}) is connected.
By 4 and the fact that Treeq' is a tree, S must not be a node in T'. So the added cost due to nodecost(S) is one. If the attribute in c is not an attribute in any clause of T', then there is an additional added cost of one to nodecost(Q), ending the proof of the inductive step. If the attribute in c is already in some clause of T', c may or may not add a cost of one to the nodecost of Q, but the conclusion holds in either case. End of proof of subclaim.

Therefore the cost of QG(Treeq') is at least ||C|| + m so K ≥ m. We complete our proof of the claim by showing that using these same m attributes, we can construct a singleton spanning tree QG(Treeq) of QG(q) of cost ||C|| + m. We do this by showing how to construct QG(Treeq) such that each edge in QG(Treeq') is a path in QG(Treeq):
(1) Start with an edge between R and every other relation T.
(2) Suppose there is an edge S.x = S'.x in QG(Treeq'). If S ≠ R and S' ≠ R, add clauses {S.x, R.x} and {S'.x, R.x} to Treeq. (We know that q has {S.x, R.x} and {S'.x, R.x} since q and q' are equivalent and {S.x, S'.x} could not be inferred from q otherwise.) If S = R or S' = R, add {S.x, S'.x} to Treeq.
(3) If an edge has more than one clause, throw out all but one of these clauses.

Since we have used the same attributes as in QG(Treeq') and since each relation in QG(Treeq) is either R or connected only to R by a singleton clause, the cost of QG(Treeq) will be ||C|| + m which is no greater than ||C|| + K. So there will be a hitting set of size K at most. (This claim has shown that a cheap spanning tree of an equivalent query implies the existence of a small hitting set. We proved the converse in the part of the proof before the claim.) □

3.2. Single-clause Tree Queries

A *single-clause tree query* q is a tree query that is equivalent to some query q' such that QG(q') = ((V,E'), q'), where (V,E') is a tree and clauses(e,q') is a singleton set for every e in E'.

Example: The query R.A = S.B, S.B = T.C, T.C = R.A, T.D = Q.E is a single-clause tree query.

Proposition 1: If q is a single-clause tree query, then QG(q^+) is singleton.

Proof: Since q is a single-clause tree query, there is a query q' equivalent to q such that QG(q') is a tree with a single clause on each edge. QG(q^+) must be a supergraph of QG(q').

Suppose R and S are not adjacent in QG(q'). Any clause associated with the edge {R, S} in q^+ must be redundant with the clauses along a path from R to S in QG(q'), since q' and q^+ are equivalent. There is only one such path since QG(q') is a tree. Moreover, every edge along this path contains one clause. So only one clause relating R and S can be redundant with the clauses along this path.

Suppose R and S are adjacent in QG(q'). Since QG(q') is a tree, there can be no path from R to S other than through {R,S} in QG(q'). Therefore no clause relating R to S could be redundant with q' -

clauses({R,S},q'). So QG(q^+) must have one clause associated with {R,S} as well. []

Corollary: If q is a single-clause tree query, then QG(q) is singleton. □

Proposition 2: If c1 is a query of k clauses without redundancies then any query c2 equivalent to c1 must have at least k clauses.

Proof: Since c1 has no redundancies, its join graph JG(c1) must have no cycles. Therefore it consists of a set of connected components which are all acyclic. Since c1 and c2 are equivalent, R.x and S.y are in the same component of JG(c1) if and only if they are in the same connected component of JG(c2). Because JG(c1) has no cycles, each connected component of JG(c1) may have no more edges than the corresponding connected component of JG(c2). Hence c2 must have at least k clauses. □

Proposition 3: If q is a tree query, QG(q^+) is singleton, and q' is a query such that QG(q') is a spanning tree of QG(q^+), then q' is equivalent to q.

Proof: Since q and q^+ are equivalent by definition, we need only show that q' and q^+ are equivalent. We proceed by contradiction. Suppose some clause R.x = S.y of q^+ is missing from q' and is not redundant with the clauses in q'. Since q^+ is singleton, this clause may only be missing from q' if there is no edge between R and S in the tree QG(q'). However, QG(q') must contain exactly one path from R to S, since QG(q') is a spanning tree of the query. Since R.x = S.y is not redundant to q', adding it to q' must not make any other clause in q' redundant. (To see this, suppose {V.q, U.w} were redundant in q' $\cup$ {{R.x, S.y}}, then there would be a cycle in the join graph, JG(q' $\cup$ {{R.x, S.y}}). The cycle would have to include {R.x, S.y} as an edge, because q' itself can have no redundancies since QG(q') is a spanning tree. But then {R.x,S.y} would also be redundant which contradicts our assumption.)

Suppose that the query is on m relations. Query q' must contain m − 1 clauses, since QG(q') is a spanning tree. Thus, QG(q' $\cup$ {{R.x, S.y}}) has m clauses. Let qextra ($\subseteq q^+$, and possibly null) be a set of additional clauses, so q' $\cup$ {{R.x, S.y}} $\cup$ qextra is equivalent to q^+ and has no redundancies. Let the resulting query be called q''. Query q'' has at least m clauses. Any query q''' that is a subset of q^+ and equivalent to q^+ must have at least m clauses too, by proposition 2. Since q''' is a subset of q^+ and QG(q^+) is singleton, every clause in q''' must relate a distinct pair of relations so QG(q''') must have at least m edges. Since QG(q''') relates m different relations, QG(q''') must contain a cycle. Therefore q^+ is not a tree query. But q^+ and q are equivalent, so q is not a tree query either. Contradiction. □

Proposition 4: Adding or removing a redundant clause from a query graph does not change the repartitioning cost of the graph.

Proof: Suppose the clause in question is {R.x, S.y}. Since it is redundant, there exist members among the other clauses of the query of the form {R.x,T_1.z_1}, {T_1.z_1,T_2.z_2}, {T_2.z_2,T_3.z_3}, ... {T_m.z_m,S.y}. Moreover, since the query graph has no self-loops (by our preprocessing) R ≠ S. Therefore, with or without {R.x, S.y}, R.x will be in a clause associated with R and similarly for S.y. So, the nodecost of R and S are the same in the new graph and the old one. □

Corollary: Any two equivalent queries have the same repartitioning cost. □

Suppose q is a single-clause tree query. By proposition 1, $QG(q^+)$ is singleton. By proposition 3, every spanning tree of $QG(q^+)$ is equivalent to q. Hence every spanning tree of QG(q) is also equivalent to q. By proposition 4, every such spanning tree has the same cost.

Given a query q that we suspect is a single-clause tree query, we do the following

```
single-clause-optimize(q: query)
begin
 construct a query graph for q;
 eliminate self-loops;
 take any spanning tree QG(q') of QG(q);
 if q' is equivalent to q
   then
          {q is a single-clause tree query}
          {q' is an optimal repartitioning
          spanning tree}
          return q'
   else
          {q is not a single-clause tree query}
          try other heuristics on q
  end if
end
```

We thus have a constructive proof of the following theorem.

Theorem 2: The minimal repartitioning problem requires linear time for single-clause tree queries. □

3.3. Processing a singleton spanning tree

Suppose we are given a (possibly non-minimum cost) singleton spanning tree query QG(q) graph of cost k plus a set of intra-relation restrictions and we want to process the query. Any direct execution of the query (i.e. one that does not transform query q to another query) requires at least k repartitionings. To see this, consider some node R in QG(q). Since QG(q) is a singleton spanning tree, if R.x appears in a clause, then R or an intermediate result containing R will have to be partitioned on x at some point in the query execution. If x is the partition attribute of R, then R may not have to be repartitioned to allow the joins associated with R.x to occur. Thus, the nodecost of R in QG(q) is in fact the minimum number of times R will have to be repartitioned during the execution of the query. The same holds for any other relation in the query.[12] So, the number of repartitionings must be at least the sum of the nodecosts, which

[12] One might think that repartitioning intermediate relations might achieve two repartitionings at the cost of one. For example if the x and y attributes are equal in some intermediate relation created by the join R.x = S.y, any repartitioning on x would also cause a repartitioning of y. This is true but irrelevant, since immediately after the join corresponding to R.x = S.y, the intermediate relation was already partitioned on those attributes.

is k.

```
construct_strategy(QG(q): singleton spanning tree)
begin
 rename the attributes of relations in q so all attributes
   of different relations are distinct;
 assign an arbitrary total ordering <R
  on the attributes of the clauses
  associated with each relation R,
  with the restriction that the partition
  attribute of R should be the minimum in the ordering;
 create a directed graph on the clauses with edges
  {({R.x, S.y}, {R.w, T.z}) | x <R w
          and {R.x, S.y} ∈ q and {R.w, T.z} ∈ q}
  this is called the ''clause graph'' CG(q, <);
 process the query in any order consistent with
  this graph
end
```

Note that the second and fourth steps both allow choices to be made. These choices may derive from independent heuristics, e.g. join smallest relations first. The restriction in the second step ensures that partition attributes of relations are not wasted.[13]

As for the total cost, the procedure requires that joins having to do with a particular attribute of a relation R are not interleaved with joins having to do with other attributes of R. Hence, each relation R will be repartitioned only as often as there are distinct attributes of R in join clauses associated with R (minus one, if R's partitioning attribute is one of these attributes). This is exactly the repartitioning cost of the QG(q). It remains to show that the fourth step can be done, i.e. that the clause graph is acyclic.

Lemma: The clause graph, CG(q,<), is acyclic for any singleton spanning query graph QG(q).

Proof: For each relation R in the query q, let the *clauses associated with* R denote the set of clauses of the form {R.x,S.y} ∈ q for some x, S, y. (Intuitively, these are clauses that some attribute of R is involved in.)

Claim: For any set Rel of relations, if there is a cycle in the clause graph CG(q,<) whose clauses all are associated with the members of Rel, then there is a cycle whose clauses are all associated with exactly one member of Rel.

Proof of Claim: By induction. If Rel consists of one relation, the claim is obvious.

Inductive step: suppose the claim is true for some set Rel not including relation S. We show it is true for Rel ∪ {S}, where S is adjacent to at least one member R of Rel in QG(q).

[13] Suppose there are several copies of a relation each partitioned on a different attribute. If at least one of those attributes is in a clause associated with R in the query, choose such an attribute as the partition attribute for the purposes of this algorithm.

Proof of inductive step. Consider figure 7. Any path from clause c (associated with R) to c′ (associated with S) in CG(q,<) must go through the clause {R.x, S.y}, because the underlying query graph QG(q) is a tree. Similarly for paths from c′ to c. Therefore any cycle in the clauses associated with Rel ∪ {S} including both c and c′ must include {R.x, S.y} and hence must have a subcycle consisting only of clauses associated with members of Rel. But such a cycle must in turn have a subcycle whose clauses are associated with one member of Rel by induction hypothesis. End of proof of claim.

By the claim, a cycle in the clause graph implies the existence of a cycle among the clauses associated with a single relation, say T. But this is impossible since $<_T$ is a strict ordering and QG(q) is singleton. □

>From the lemma and previous discussion, we have given a constructive proof of the following.

Theorem 4: Given a spanning tree of cost k, we can process the query using k repartitionings. □

4. Loose Ends

When one relation of a join is much smaller than another, repartitioning the larger relation may not be necessary. For example, suppose our query consists of a single join R.A = S.B. Suppose that both relations are partitioned on their A attributes, but the current instance r of R is much smaller than (i.e. by a factor of n, the number of processors) than the current instance s of S. Instead of repartitioning S and then performing the join, we might consider broadcasting all the tuples of R to **every** processing node. This might save time in the join, but it would leave S partitioned on its A attribute rather than its B attribute. Thus, if S is to be partitioned on its B attribute eventually anyway (for a more complex query), this ‘‘copying technique’’ may be the wrong choice. Whether this technique is even locally (i.e. to the join in question) profitable may only be apparent during the processing of a query as it may depend on the size of intermediate results. Similar considerations would apply if we decided to process this join as a pair of semi-joins: send S.B values to respective processing nodes, then have those that match R tuples return.

We have not discussed the possibility of using multiple copies of relations, perhaps partitioned on different attributes. The cost function per node changes in a query q: Let Att(R,q) = {R.x | ∃ S, ∃ y such that {R,S} ∈ E, R ≠ S {R.x, S.y} ∈ Clauses({R,S},q)}. $\text{Nodecost}_{E,q}(R) = || \text{ Att}(R,q) ||$ - (if Att(R,q) contains a partitioning attribute then 1 else 0). In the example of figure 2, if one copy of R is partitioned on D, one copy of T on C, one copy of S on A, and another copy of S on B, the cost would be one. It would not be zero, since we would have to choose which copy of S to use. Intuitively, we have more partition attributes to choose from, so the chances of the nodecost of R being one less than Att(R,q) are good. However, we can only use one of the copies and hence only one partition attribute for one relation in the query, so the cost is Att(R,q) – 1 at best. (Of course, if we have Emp1.mgr = Emp2.name, we can use one copy of the employee relation for emp1 and another for emp2.)

Our results have all assumed that the relations were partitioned on a single attribute. This is usually advantageous. For example, consider a join between R and S based on the clauses {{R.A = S.C}, {R.B = S.D}} where R is partitioned on A and S on C. One can process the join based on {R.A = S.C}, saving {R.B = S.D} as an intra-relation restriction on the result. On the other hand, if the join were {R.A = S.C}

and R were partitioned on AB and S were partitioned on CD, both would have to be repartitioned to A and C respectively. This example suggests how to handle multi-attribute partitions. Treat the partition attributes (e.g. AB) as if they were a single attribute distinct from their constituents (e.g. A or B alone) attributes but as a constituent of any supersets (e.g. ABC). Thus, to perform {{R.A = S.C}, {R.B = S.D}, {R.E = S.F}} given AB and CD as partition attributes, process {{R.A = S.C}, {R.B = S.D}} and save {R.E = S.F} as an intra-relation restriction.

5. Conclusion

Given a symmetric parallel network with high bandwidth, optimal speedup is attainable for many functions. In the range where this network assumption holds, our paper proposes a framework for query optimization. The framework assumes that relations are horizontally partitioned and that the partitioning is based on a subset of the attributes. (That is, all tuples with the same values of those attributes should be in the same processing node.) The framework further assumes that repartitioning is an expensive operation.

The main result is that heuristics will have to be used to minimize the number of repartitionings of general queries. For a certain class of tree queries (ones with single clauses along each query graph edge), the problem can be solved in linear time.

Acknowledgements

Many thanks to Richard Cole, Zvi Kedem, Bud Mishra, and Paul Spirakis for helpful discussions concerning this paper, and to Boris Aronov and Vladimir Lanin for critical comments.

REFERENCES

[AHY83] P. M. G. Apers, A. R. Hevner, and S. B. Yao, "Optimization Algorithms for Distributed Queries" IEEE Transactions on Software enginnering, vol. SE-9, no. 1, January 1983.

[B79] E. Babb, "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS 4,1 (March, 1979), 1-29.

[B84] K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," Proceedings of the 1984 very large data base conference.

[BC81] P. A. Bernstein and D. W. Chiu, "Using Semijoins to Solve Relational Queries," J. ACM 28:1, pp. 25-40, 1981.

[BDFW83] - H. Boral, D. J. DeWitt, D. Friedland, and W. K. Wilkinson, "Parallel Algorithms for the execution of relational database operations," ACM Transactions on Database Systems vol. 8, no. 3, September 1983, pp. 324-353.

[BHK79] J. Banerjee, D. K. Hsiao, and K. Kannan, "DBC - A Database Computer for Very Large Databases," IEEE Trans. Computers, vol. C-28, No. 6, June 1979.

[BGWRR81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie Jr. "Query Processing in a System for Distributed Databases" (SDD-1). ACM Trans. Database Syst. 6, 4 pp. 602-625, 1981.

[BO79] B. Berra and E. Oliver "The role of associative array processors in database machine architectures" IEEE Computer, 12, 3, 53-61, 1979.

[BR85] H. Boral and S. Redfield, "Database Machine Morphology" Proceedings of the 11th int. conf. on very large data bases pp. 59 - 71, 1985.

[CH80] - D. M. Chiu and Y.C. Ho, "A methodology for interpreting tree queries into optimal semi-join expressions" ACM SIGMOD conference 1980, pp. 169-178.

[CH82] W. W. Chu and P. Hurley, "Optimal Query Processing for Distributed Database Systems" IEEE Trans. Computing, C-31, 9, pp. 835-850, 1982.

[D79] D. J. DeWitt, "DIRECT -- a multiprocessor organization for supporting relational database management systems," IEEE Transactions on Computers, C-28, 6, 1979.

[DG85] D. DeWitt and R. Gerber, "Multiprocessor Hash-based Join Algorithms" Proceedings of the 11th int. conf. on very large data bases pp. 151 - 164, 1985.

[DKOSSW84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, D. Wood, "Implementation Techniques for Main Memory Database Systems" SIGMOD record, vol. 14, number 2, June, 1984, Boston, Mass., pp. 1-8.

[ES80] R. S. Epstein and M. Stonebraker, "Analysis of query processing strategies for distributed database systems." sixth international conference on very large databases, Montreal, October, 1980.

[G79] - M. H. Graham, "On the universal relation," Tech. Report, University of Toronto, Toronto, Ontario, Canada, September 1979.

[GGKARS83] - A. Gottlieb, R. Grishman, C. K. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing a MIMD, shared-memory parallel computer" IEEE Trans. on Computers C-32 pp. 175-189, 1983.

[GJ79] - M. R. Garey and D. S. Johnson, *Computers and Intractability -- a guide to the theory of NP-completeness* W. H. Freeman and Company, San Francisco, California, 1979.

[GS81] J. R. Goodman and C. H. Sequin, "HYPERTREE: a multiprocessor interconnection topology," IEEE Transactions on Computing, 30, 12, 1981.

[GS82] B. Gavish and A. Segev, "Query Optimization in Distributed Computer Systems" In Management of Distributed Data Processing, J. Akoka, Ed. Elsevier North-Holland, New York, pp. 233-252, 1982.

[GoodShmu82] - N. Goodman and O. Shmueli, "Tree queries: A simple class of relational queries" ACM Transactions of Database Systems vol. 7, no. 4, December 1982, pp. 653-677.

[H83] D. K. Hsiao, editor *Advanced Database Machine Architecture* Prentice-Hall Inc., 1983, pp. 203-255.

[HY79] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems" IEEE Trans. Softw. Eng. SE-5, 3, pp. 177-187, 1979.

[HY78] A. R. Hevner and S. B. Yao, "Query Processing on a Distributed Database" Proceedings Third Workshop on Distributed Data Management and Computer Networks, August 1978, pp. 91-107.

[HSN85] B. K. Hillyer, D. E. Shaw, and A. Nigam, "NON-VON's Performance on Certain Database Benchmarks" IEEE Transactions on Software Engineering, to appear.

[JK84] M. Jarke and J. Koch, "Query Optimization in Database Systems" ACM Computing Surveys, vol. 16, no. 2, June 1984, pp. 111-152.

[KTM84] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Architecture and Performance of Relational Algebra Machine Grace" IEEE Parallel Processing Conference 1984.

[KTM83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Grace: Relational Algebra Machine Based on Hash and Sort -- Its Design Concepts" Journal of Information Processing, vol 6, no. 3, 1983.

[MH81] M. J. Menon and D. K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI," Proceedings International Conference on Very Large Database, 1981.

[PBGHKMMNW85] - G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture," Proc. IEEE 1985 conference on parallel processing, pp. 764-771.

[O82] E. A. Ozkarahan 1982, RAP "Database Machine/Computer Based Distributed Databases." In Proceedings of the 2nd International Symposium on Distributed Databases. (Berlin, FRG). Elsevier North-Holland, New York, pp. 61-80.

[S79] S. Y. W. Su 1979 "Cellular-logic Devices: Concepts and Applications" IEEE Computer 12, 3, 11-25.

[Schk82] M. Schkolnick, "Physical database design techniques." In *Data Base Design Techniques II* S. B. Yao and T. L. Kunii, Eds., Springer-Verlag, pp. 229-252, 1982.

[Schm79] J. W. Schmidt, "Parallel processing of relations: a single-assignment approach." In proceedings of the IEEE 5th international conference on very large data bases, pp. 398-408, 1979.

[SL75] S. Y. W. Su and G. Lipkovsky 1975, "CASSM: A Cellular System for Very Large Databases" In Proceedings of the 1st International Conference on Very Large Data Bases" Framingham, Mass., Sept. 22-24. ACM, New York, pp. 456-472.

[SS85] D. Shasha and P. Spirakis, "Join Processing in a Symmetric Parallel Environment" NYU TR #158, March, 1985.

[SZ84] R. K. Shultz and R. J. Zingg, "Response Time Analysis of Multiprocessor Computers for Database Support" ACM Transactions of Database Systems, vol. 9, no. 1, March, 1984, pp. 100-132.

[T84] Y. Tanaka, "MPDC: Massive Parallel Architecture for Very Large Databases" Proc of International Conference on Fifth Generation Computer Systems pp. 113-138, 1984

[VG84] - P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine" ACM Transactions of Database Systems, vol. 9, no. 1, March, 1984, pp. 133-161.

[WY76] E. Wong and K. Youssefi, "Decomposition -- a strategy for query processing" ACM TODS 1, 3 Sept. 1976, pp. 223-241.

[YO79] - C. T. Yu and M. X. Ozsoyoglu, "An algorithm for tree-query membership of a distributed query" In proc. Compsac79, IEEE, New York, Nov. 1979.