A Model of Query Performance in Dynamic Distributed Federated Databases Taking Account of Network Topology

Akira Kawaguchi
Abbe Mowshowitz
The City College and the CUNY Graduate Center
USA

Andrew Nagel
Andi Toce
The CUNY Graduate Center
USA

Graham Bent
Patrick Dantressangle
IBM UK

Abstract—We examine the problem of distributed join query optimization in a Dynamic Distributed Federated Database (DDFD) and the role that network topology, specifically an engineered hypercube, could play in the cost estimation. Considering that many nodes in a DDFD may rely on batteries and wireless radios, we should like to minimize network information transfer to maximize network availability and lifetime. While in any topology it may be assumed a querying node can estimate distance to data sources, in an unknown topology, the querying node has no direct knowledge of inter-site distances. In a hypercube, these distances can be easily calculated at the querying node based on the sites’ labels. What is more, query optimization in the past has not fully taken account of the topology of networks as a cost factor in executing joins.

Due to the unavailability of open-source implementations, we have developed and implemented a distributed join query optimizer for comparative performance studies. Our preliminary experiments show that incorporating inter-node distances into the cost model often produces much better plans, regardless of the search algorithm (greedy or dynamic programming). We confirm a significant advantage of semijoin adoption and quantify the performance of the greedy algorithm in relation to dynamic programming, but acknowledge that a dynamic programming approach may often be infeasible in a DDFD for large-scale distribution. Based on our findings, we will present some practical considerations for improvement.

Index Terms—query performance, dynamic, distributed, federated, database, network topology, join, semijoin

I. INTRODUCTION

The relational model proposed by EF Codd in 1970 effectively decoupled specification of requests for data from the actual methods used to store the data and process those requests. Database management systems that adhere to the relational model bear the responsibility of parsing high level user requests and assembling the result. But there are many distinct sequences of relational operations that produce an identical result set and differ in cost of execution, sometimes by several orders of magnitude. System R was developed in part to demonstrate that the process of generating and choosing an optimal plan could be automated effectively [4]. Distributed query optimization is a rich topic extending to the earliest days of database technology. The methods developed for System R in the 80’s continue to be the most widely used in commercial products. But evolution and diversification of database architecture since the 90’s has exposed the need for more sophisticated query optimizers. This is particularly true for distributed, heterogeneous, dynamic and/or federated databases.

The plan search space of a query consists of the set of all execution plans satisfying the query that the query optimizer must explore. The optimization procedure consists in enumerating or generating plans, estimating their respective costs, and then choosing the cheapest plan. The first two tasks are far from trivial. The plan space grows at an enormous rate with query size and other factors. Estimating costs is inherently non-trivial since statistically based methods may introduce errors in estimates of intermediate join results. Nevertheless, absent accurate cost estimates, optimization is meaningless. These difficulties are compounded in a Dynamic Distributed Federated Database (DDFD). The plan space may grow even larger by taking account of replication or fragmentation of data. The location of intermediate results must be considered. Local cost models, statistics on remote relations, and communication costs may be unknown, and the optimizer may incur costs obtaining this information.

We focus on DDFD query optimization in an engineered hypercube network as presented in [1]. We believe network costs to be most significant in this environment, assuming that network life and availability should be maximized, and this is our main focus. Note however that the cost models we use could be extended easily to include other factors. For purposes of minimizing networks costs we validate the use of semijoin operations as reducers of data transmission. A key element in our investigation is the exploitation of knowledge of the hypercube structure to facilitate optimization. We can use it to provide consolidated cost estimates for logical tables, which are ultimately fragmented relations, thus decreasing the plan search space. Most importantly, we consider internode network distances and the role of topology. In a general dynamic distributed database, knowledge of these distances might require $O(n^2)$ messages as nodes query each other. In a hypercube network, these inter-node distances require no extra messages, since they can be calculated locally using node
The main contribution of this research is a quantitative evaluation of distributed query performance that takes account of network topology. We will show that knowledge of internode distances often produces much better plans than if network costs were assumed to be uniform. Verified by experiment, this finding is of potential practical value in military environments where the efficiency of distributed query optimization is critical. The rest of this paper is organized as follows: Section II begins with problem characteristics and Section III illustrates one example to base our solution approach and research significance. Section IV presents the theoretical aspect of the problem solution. Section V presents our implementation of distributed join query optimizer and experimental performance analysis. We briefly cover planned future work and conclude our work in Section VI.

II. BACKGROUND

The size of the plan search space is what makes query optimization NP-hard. Plans are usually represented as a tree whose inner nodes are operators like select or join that represent the result of applying that operation on its child nodes. The leaf nodes represent access to a base relation while the root of this tree represents a view of the data corresponding to the user’s query. Some techniques are used almost universally to decrease the search space, sometimes in a query pre-processing phase. For instance, selection and projection should, if possible, always occur before a join since they are local operations and can only reduce the size of the relation that needs to be joined. For this reason, the literature focuses on optimizing the ordering of join operations. Given a join on n relations, even if we only consider so called ‘linear joins’ as in Figure 1 there are n! possible join orders due to commutativity and associativity of joins. There are even more possible join orders if we consider ‘bushy’ plans as in Figure 2. Bushy plans might result in lower network cost and allow for parallelization in execution.

Query optimizers differ in how much of the search space they explore and in the relative quality of the plans generated. Classical methods generally fall into dynamic programming based or greedy algorithms, each representing a trade off in complexity and quality.

A. Dynamic Programming

System R implemented the earliest optimizer using a bottom up dynamic programming algorithm. The optimizer only considers linear join plans and uses the principle of optimality to prune non-optimal sub-plans, reducing the number of join orders considered to $O(n2^{n-1})$ [5]. This algorithm guarantees finding the optimal (linear) join plan for the given cost model. The principle of optimality simply states that optimal plans are composed of optimal sub-plans. A description of this process and its application to several optimization problems is given in [6]. The bottom up method first considers all possible 2-way joins, then all possible 3-way joins are generated by joining another relation to the 2-way joins. If at any point, two plans are found that generate the same result, the plan with higher cost can be discarded, speeding up the search.

But the search space is still exponential, and dynamic programming can be infeasible for queries joining large numbers of tables, or in distributed settings where data is replicated and/or fragmented. Another consideration in distributed databases stems from the location of the result set. While in a centralized system, the result of A $\bowtie$ B and B $\bowtie$ A may be considered identical, in a distributed system the result will be generated at one site or the other. One of these may be closer to other required data sources, and ultimately participate in the optimal plan, so both options may need to be considered.

B. Greedy Methods

Not long after development of System R, researchers became interested in optimizing queries in distributed databases. One of the earliest proposed algorithms was SDD-1 [3], which like our approach, seeks to minimize network transfer. It uses the heuristic of choosing to do all data reductions like selections and projections first. But when none are left, it iteratively chooses the join that requires sending the least data. There is no consideration of the resultant cardinality, selectivity, or location until the next round, and there is no backtracking. SDD-1 notably uses semijoin operators to achieve joins, which putatively reduces overall data transfer [16].

The attractiveness of SDD-1 lies in its ease of implementation and more so in its efficiency. It will return a query plan using polynomial time and space in the number of joins. However, it does not produce good enough plans. While for some queries it may find the optimal plan, the likelihood decreases with query size. The other big drawback of SDD-1 is that it does not consider network topology; rather, network cost is based strictly on the number of bytes transmitted. This aspect of the cost model can be extended easily as we have done in our experiments with greedy methods.

C. Hypercube Topology

A hypercube with $n = 2^k$ nodes is said to be of dimension $k = \log(n)$. Each node has a k bit label and two nodes are adjacent if their labels differ in exactly one position. Separate arguments can be made for the value of a hypercube network topology regarding robustness and message routing [17]–[19], but we specifically exploit the labeling system to improve query optimization. A querying node in a DDFD that is going to attempt optimizing the query plan must identify the nodes with relevant data. We assume this is done with a
flood query, and all sites that have a copy of a referenced relation respond with a message including the site’s label and the relation’s cardinality and selectivity statistics. Distribution does not fundamentally alter the optimization process, but simply changes the cost model used, potentially increases the number of relations queried, and potentially increases the number of copies of relations. These increases contribute exponentially to the size of the plan search space.

The optimizing node will use the labels of the data sources to calculate inter-node distances or hop counts. So instead of assuming all network weights to be 1, as SDD-1 did, data transfer is weighted by path length. For this to be achieved in an unknown topology, the optimizing node would have to form a list of nodes that might participate in query execution, then send this list to each of those sites. This would be followed by the exchange of pairwise messages to find inter-node distances, which are returned to the querying node. If there are $n$ sites that might participate, $O(n^2)$ messages would have to be sent before optimization could even begin. This might be worth doing since using the inter-node distances in the cost model offers great potential for cost savings. But in a hypercube, this knowledge is gained almost ‘free.’

**D. Logical Tables**

Labels can be used to improve cost estimation by calculating inter-node distances in any hypercube embedded database management system, but further exploitation is possible in DDFDs that implement the logical table concept. A logical table is regarded by the database user as one relation, but it may actually be partitioned across many nodes. If we consider only the logical table when enumerating join plans and not all the sites, we dramatically reduce the search space. This necessitates determining a cost model for operations on the logical table. It is simple enough to apply the usual cardinality and selectivity based cost estimates by using the sum of fragment cardinalities and an average of the fragment selectivities, weighted by their respective cardinalities. This is sufficient if the cost model is going to treat all network distances as equal.

But if we have incorporated network distance into our cost model, we need some metric to estimate ‘distance’ of this logical table to other nodes, even though this table is dispersed over multiple location. A hypercube topology makes this estimation possible and relatively simple. We assign one node as the logical table’s center. The center need not actually store any piece of the logical table, and does not necessarily need to know that it has been so designated. The center only exists to give us a point from which to calculate distances to other nodes. Only the querying node that is actually optimizing the query needs to know the center for cost estimation, and it can calculate this locally using only the contributing nodes’ labels.

**III. MOTIVATING EXAMPLE**

Consider a federated process of data analysis that needs to gather and combine several items contained in three relations $R_1$, $R_2$, and $R_3$. Each of the relations is placed in a distinct node in a network whose topology is known. This relation could be a concrete instance or temporary materialization that resulted from some localized operation at the hosting node. Suppose also that the pairwise node distances are known, namely, $dist(R_1, R_2) = 2$, $dist(R_1, R_3) = 8$, and $dist(R_2, R_3) = 8$. To complete the data analysis it is necessary to obtain the results from a distributed three-way (natural) join of $R_1 \bowtie R_2 \bowtie R_3$, and we aim to do this in the most cost-effective way.

Figure 3 shows the data properties for each of the three relations. This kind of information is relatively easily obtained in most database systems. For instance, $R_1$ has 1,125 tuples of size 1,074 (bytes), in which column $A$ (132 bytes) and $B$ (203 bytes) will be matched with $R_3$ and column $C$ (64 bytes) will be matched with $R_2$ whereas other columns denoted $\alpha_1$ (677 bytes) will not participate in the join but need to appear in the final result. Columns $A$, $B$, and $C$ hold 201, 434, and 158 distinct values respectively. To cope with variable-length data types, the sizes of tuples and columns are expressed as averages. Join columns may not have the same length.

Given an empirical observation that the communication cost between any two nodes is proportional to their distance, minimizing the overall data transmission cost by accounting for the distance to access remote relations is a viable approach. For now, let us consider the strategy of sending out whole relations, not their parts. One can think of a simple greedy method to estimate the data transmission overhead by inspecting all probable query plans—in this case, pairwise joins such as $(R_1 \bowtie R_2) \bowtie R_3$ and $R_1 \bowtie (R_2 \bowtie R_3)$. Planning the execution of $R_1 \bowtie R_2$ requires comparison of the cost of sending $R_1 \rightarrow R_2$ with $R_2 \rightarrow R_1$ to choose the smaller one. This estimate must incorporate network distance as a factor. In this example, $R_1 \rightarrow R_2$ is estimated to be 1.2MB $\times$ 2 and $R_2 \rightarrow R_1$ is 375KB $\times$ 2. Therefore, $R_2$ is sent to $R_1$’s node for execution. Similarly, the best cost of executing $(R_2 \bowtie R_3)$ is 3MB $\times$ 8 by sending $R_2 \rightarrow R_3$. Therefore, the greedy method takes $(R_1 \bowtie R_2) \bowtie R_3$ as a promising direction.

For planning the third join with $R_3$, the size of the join of $R_1$ and $R_2$ needs to be estimated. Much work has been done to derive the estimates of join size. A variant of the value matching probability method [9] is applicable to the collection.
of cardinality and selectivity statistics which must be known 
a priori or gathered during data discovery. The number of
tuples produced by \( R_1 \bowtie R_2 \) is estimated to be \( 1,125 \times 872/max(158,353) \approx 2,779 \), which is used to compute an
estimated total size of the join result. The cost of sending \( R_3 \)
to \( R_1 \)'s node (10.9MB) instead of sending the result in \( R_1 \)'s
node to \( R_3 \)'s node (33.3MB) is significantly smaller. The data
transmission strategy for \( (R_1 \bowtie R_2) \bowtie R_3 = R_2 \rightarrow R_3 \)
and \( R_3 \rightarrow R_3 \) with a total transmission cost of 11.7MB with
the final result at node 1.

Note that the greedy method never revisits the de-selected
plan of \( R_1 \bowtie (R_2 \bowtie R_3) \). It starts at the lowest level
of the evaluation tree, chooses the most promising pair to
climb up the tree for the next level’s evaluation (thus called a
hill-climbing method). Furthermore, an \( n \)-way join often pro-
duces significantly different intermediate contents depending
on the order in which sub-joins were performed. Therefore,
the execution plan needs to inspect half of all possible join
permutations, such as

\[
R_1 \bowtie R_2 \bowtie R_3 \quad R_1 \bowtie R_3 \bowtie R_2 \quad R_2 \bowtie R_1 \bowtie R_3
\]

In this example, the first case and the third case produce the
same result while the second case ends up with an inferior
plan with a total transmission cost of 12.7MB.

A. Significance of This Research

As the previous example indicates, efficient execution of
the \( 3 \)-way join depends on the amount of information in
each node and the distance between nodes. Achieving such
efficiency is facilitated by carrying out the process in a
network with a topology that allows for easy computation of
internode distances. Some questions that arise in this context
are examined below.

First, classic query optimization procedures assume there
is no cost associated with obtaining knowledge of internode
distances. We show that better plans can be found with the
use of internode distances than without, and if the actual
cost of obtaining distance information is low as in hypercube
networks, the cost of obtaining such plans is justified. Consider
the above example. If all the distances were assumed to be 1,
the best execution plan produced by the same procedure would
be different, namely, \( (R_2 \bowtie R_3) \bowtie R_1 \) with data transmission
strategy of \( R_2 \rightarrow R_3 \) followed by \( R_3 \rightarrow R_1 \). If such a
plan is inattentively executed in the original network, the total
data transmission cost becomes equivalent to the inferior plan
having 12.7MB transmission cost, thereby consuming close to
10% of additional network bandwidth. Clearly, the lowest
possible cost algorithm for executing multinode joins will
exploit knowledge of such internode distances.

Second, our discussion thus far only considered data ex-
change based on whole joins, i.e., sending out entire relation
operands between nodes. Instead, a semijoin operation \([\text{2}]
\) can reduce the data transmission cost by sending out only
join-relevant columns from one to another, and executing a
smaller join at the sent site. Semijoins however induce more
operations but possibly on smaller operands \([\text{13}] \). Many of
the early studies pointed out the trade-off by minimizing the
communication cost at the expense of the local processing
cost. However, the scale of distribution and the power of local
processing differs, perhaps more than an order of magnitude
today. This issue must be re-examined from the perspective of
a known network topology. For instance, semijoin use as
discussed later for the above example produces an execution
plan with a cost of 6MB or 48% less than without semijoins.
The knowledge of network distances gives a solid foundation
in deciding the choice of whole join or semijoin.

Third, the greedy method often concludes with a local
optimum. There seems no published work that attempts to
define cost-benefit models and their implementation based on
rules for generating distributed join execution plans to obtain
the ultimate best. One of the main reasons for undertaking our
current research is to investigate ways of obtaining efficient
query optimization schemes. Dynamic programming allows
for finding a global scheme thus providing a check on the
local maximum produced by the greedy method. We will see
later that the cost of the optimal semijoin incorporated plan for
the above example is 3.2MB, 62% less than the one produced
by the greedy method. The limit of such a mechanism also
needs to be investigated.

IV. OPTIMAL JOIN SCHEDULE

Consider the distributed processing of \( n \)-way joins for the
collection of relations \( R_1 \) through \( R_n \) \((n > 2)\), each of which
resides at a distinct node in a network. The optimal cost of
the join is formulated by the use of dynamic programming.

Definition 4.1 (Join Selectivity): Let \( \text{card}(R_i) \) denote the
tuple count of relation \( R_i \); let \( A_{i,j} \) be an attribute set specified
as a matching condition in a natural join operation of two
relations \( R_i \) and \( R_j \), in which the attribute set \( A_i \) of \( R_i \)
is matched with the attribute set \( A_j \) of \( R_j \) for their value
equivalence (note \( A_{i,j} = A_i \cup A_j \)). We use the notation
\( R_i \bowtie A_{i,j} \bowtie R_j \) to express the natural join of \( R_i \) and \( R_2 \). The
join selectivity of this natural join is defined as

\[
\text{join selectivity} = \frac{\text{card}(R_i \bowtie A_{i,j} \bowtie R_j)}{\text{card}(R_i) \cdot \text{card}(R_j)}
\]

Numerous approaches have been proposed to estimate join
selectivity. However, choosing the best method depends on the
value properties of \( A_{i,j} \) and such considerations are beyond the
scope of this paper. We will use the join selectivity approxi-
mation given in \([\text{9}]\). Let \( A_i = \{a^1_i, a^2_i, ..., a^k_i\} \) and \( A_j = \{a^1_j, a^2_j, ..., a^k_j\} \), so that the natural join can be expressed
more precisely as \( R_i \bowtie A_{i,j} \bowtie R_j \). Let \( vcount(a^m_i) \) be the number of distinct values in the attribute
\( a^m_i \) and similarly \( \text{vcount}(a^m_j) \) be the number of distinct values in
the attribute \( a^m_j \) where \( 1 \leq m \leq k \). Then, the join selectivity
is approximated by computing

\[
1/\max(vcount(a^1_i), vcount(a^1_j)) \times \cdots \times 1/\max(vcount(a^k_i), vcount(a^k_j))
\]

For example, suppose \( R_1 \) and \( R_2 \) represent educational
records in a university. \( R_1 \{\text{sid, course, semester}\} \)
holds a student registration history for the past ten semesters and \( R_2 \{ \text{ iid, course, semester} \} \) holds an instructional history of a certain department for the past four semesters. In every semester, each of 250 students registers for 4 courses among 200 course offerings, and the department offers 30 courses made by its 15 instructors. Over the past ten semesters \( R_1 \) has 10,000 records and \( R_2 \) has 120 records. Then, the selectivity estimate for \( R_1 \bowtie_{A_{1,i}} R_2 \) is \( 1/\max(200,30) \times 1/\max(10,4) = 0.0005 \). Therefore, the join is expected to produce 10000 \times 120 \times 0.0005 = 600 \) tuples.

One way of handling \( R_i \bowtie_{A_{i,j}} R_j \) is to send a whole relation instance from the node hosting a smaller size relation to the other site, so that the join can be processed at the sent node. Let \( \text{dist}(R_i, R_j) \) be a network distance between \( R_i \) and \( R_j \), and let \( ||R_i|| \) be an average tuple size for \( R_i \). In a hypercube network, \( \text{dist}(R_i, R_j) = \text{sum\_bits}(\text{label}(R_i) \bowtie \text{label}(R_j)) \) in which \( \text{label}() \) is a label extraction operator for the node holding an operand relation and \( \bowtie \) is an exclusive-or operator. If \( R_i \) is smaller, the transmission cost is \( \text{card}(R_j) \cdot ||R_i|| \cdot \text{dist}(R_i, R_j) \). In addition, \( R_i \bowtie_{A_{i,j}} R_j \) can be computed by replacing one or both operand relations by a semijoin with the other relation using the following rules [2], [15]:

\[
R_i \bowtie_{A_{i,j}} R_j = (R_i \bowtie_{A_{i,j}} R_j) \bowtie_{A_{i,j}} R_j \quad (3)
\]

\[
R_i \bowtie_{A_{i,j}} (R_j \bowtie_{A_{i,j}} R_i) \quad (4)
\]

\[
(R_i \bowtie_{A_{i,j}} R_j) \bowtie_{A_{i,j}} (R_j \bowtie_{A_{i,j}} R_i) \quad (5)
\]

Eq (3) triggers work from \( R_j \)'s site in that it sends out a projection \( \Pi_{A_{i,j}}(R_j) \) to \( R_i \)'s site. Subsequently \( R_i \)'s site applies the semijoin and sends back the result to \( R_j \)'s site, so that \( R_j \)'s site can complete the join operation with the reduced instance of \( R_i \) it receives. Note that \( R_j \)'s site owns the final result. Eq (4) triggers work from \( R_i \)'s site instead but the process is effectively the same by interchanging the sites of \( R_i \) and \( R_j \) of Eq (3). Eq (5) implies a mutual exchange of \( \Pi_{A_{i,j}}(R_i) \) and \( \Pi_{A_{i,j}}(R_j) \) between the two sites. Each site can reduce its own instance applied by the semijoin with the received pair, which can be mutually exchanged back again to derive a final join result. The entire work can be processed in parallel, and both sites can have the join result in return for the increased amount of data transfer. Since only one site is required to hold the join result, Eq (5) would be eliminated from the distributed query optimization standpoint.

**Definition 4.2 (Network Cost):** Let \( ||\Pi_{A_{i,j}}(R_i)|| \) be an average size of projection \( A_i \) of \( R_i \). Let \( \varphi_{A_{i,j}}(R_i, R_j) \) be an estimate of the join selectivity of \( R_i \bowtie_{A_{i,j}} R_j \). The network cost of this join operation, denoted \( \text{cost}(R_i, R_j) \), is the smallest among the next four computations:

1. \( \text{card}(R_j) \cdot ||\Pi_{A_i}(R_i)|| + \text{card}(R_i) \cdot \text{card}(R_j) \cdot \varphi_{A_{i,j}}(R_i, R_j) \cdot ||R_i|| \cdot \text{dist}(R_i, R_j) \)
2. \( \text{card}(R_i) \cdot ||\Pi_{A_i}(R_i)|| + \text{card}(R_i) \cdot \text{card}(R_j) \cdot \varphi_{A_{i,j}}(R_i, R_j) \cdot ||R_j|| \cdot \text{dist}(R_i, R_j) \)
3. \( \text{card}(R_i) \cdot ||R_j|| \cdot \text{dist}(R_i, R_j) \)
4. \( \text{card}(R_j) \cdot ||R_j|| \cdot \text{dist}(R_i, R_j) \)

### A. Optimal Join with Dynamic Programming

Consider a certain ordering of the \( n \)‐way join \( R_1 \bowtie_{A_1} R_2 \bowtie_{A_2} \cdots \bowtie_{A_{n-1}} R_n \). In particular for \( 1 \leq i \leq j \leq n \), we consider a minimum data transmission cost, \( C(i, j) \), of computing subproblems \( R_i \bowtie_{A_{i, j+1}} R_{i+1} \bowtie_{A_{i+1, j+1}} \cdots \bowtie_{A_{j-1, j}} R_j \). The smallest subproblem occurs when \( i = j \), in which case there is nothing to compute, thus \( C(i, i) = 0 \). For \( j > i \), consider the optimal subtree for \( C(i, j) \). The first branch in this subtree, the one at the top, will split the joins in two pieces, of the form \( R_i \bowtie_{A_i} R_k \bowtie_{A_k} R_{k+1} \bowtie_{A_{k+1}} \cdots \bowtie_{A_{j-1}} R_j \), for some \( k \) between \( i \) and \( j \). The cost of the subtree is then the cost of these two partial joins, plus the cost of the final join of the results.

Let \( R_{i,k} \) and \( R_{k+1,j} \) be the result of \( R_i \bowtie_{A_i} R_k \bowtie_{A_k} R_{k+1} \bowtie_{A_{k+1}} \cdots \bowtie_{A_{j-1}} R_j \), respectively. Since \( R_{i,k} \) and \( R_{k+1,j} \) are found somewhere in the hypercube network, the total cost can be expressed by the next recursive formula:

\[
C(i, j) = \min_{i \leq k < j} \{ C(i, k) + C(k + 1, j) + \text{cost}(R_{i,k}, R_{k+1,j}) \} \quad (6)
\]

The problem to solve is complex as the solution is sensitive to the join sequence. The following is a formalization of the dynamic programming procedure.

**Algorithm 1 Optimal \( n \)‐way Join Algorithm**

1. **input:** \( n \)‐way join on relations \( R_1, \ldots, R_n \)
2. **output:** a plan \( P_{opt} \) for optimal work and its cost \( C_{opt} \)
3. **assumption:** \( \text{cost}(R_i, R_j) \) in Definition 4.2 is computed for any two relations \( R_i \) and \( R_j \)
4. **Construct \( n \)‐way join sequence set \( J \)
5. \( P_{opt} := \text{null}; C_{opt} := \infty \)
6. for all sequence \( \epsilon J \) do
7. for \( i := 1 \) to \( n \) do
8. \( C(i, i) := 0 \)
9. end for
10. for \( s := 1 \) to \( n - 1 \) do
11. for \( i := 1 \) to \( n - s \) do
12. \( j := j + s \)
13. \( C(i, j) := \min \{ C(i, k) + C(k + 1, j) + \text{cost}(R_{i,k}, R_{k+1,j}) : i \leq k < j \} \)
14. end for
15. end for
16. if \( C(1, n) < C_{opt} \) then
17. \( P_{opt} := \text{sequence} \)
18. \( C_{opt} := C(1, n) \)
19. end if
20. end for
21. **return** \( P_{opt} \) and \( C_{opt} \)

Algorithm 1 finds a join permutation with optimal cost, but it does not record join associations. For actual implementation, a bottom-up approach (or memorization) has a practical advantage over the above recursive method since it eliminates
most of redundant cost derivations as well as recording the exact associations to perform joins.

Recall that the smallest transmission cost in the example discussed in Section III is 3.2MB. Algorithm 1 produces \((R_3 \bowtie R_1 \bowtie R_1, A \bowtie R_3, A \bowtie R_1, B \bowtie R_3, B \bowtie R_3)\) as the optimal plan. The breakdown of this execution is as follows: Projection \(\Pi_{A,B} R_1\) is sent to \(R_3\)’s site to execute \(R_1 \bowtie R_1, A \bowtie R_3, A \bowtie R_1, B \bowtie R_3, B \bowtie R_3\). This result is sent back to \(R_1\) (cost 3.1MB), which allows for producing \(R_3 \bowtie R_1\) into \(R(3, 1)\) at \(R_1\)’s site. Again, projection \(\Pi_{C,R} R(3, 1)\) is sent from \(R_1\)’s site to \(R_2\)’s site to execute \(\Pi_{C,R} R(3, 1) \bowtie R_2\), which is sent back to \(R_1\)’s site to yield the final result \((\Pi_{C,R} R(3, 1) \bowtie R_2) \bowtie R_1\). This cost is only 32KB.

V. IMPLEMENTATION AND EXPERIMENT

Due to the unavailability of open-source implementations or readily executable procedures, we implemented a distributed join query optimizer as a tool for examining query performance. In this section we describe the application of the query optimizer and present preliminary performance results obtained from simulation experiments with this system.

A. Implementation

The distributed join query optimizer consists of about 3K lines of PHP code and a few system-programmed modules to realize portability among Windows, Linux, and OS X environments. It runs on a typical AMP (Apache-MySQL-PHP) solution stack. We realized the system as a pure server-side service (rather than JavaScript-heavy client) based on our desire to split the development later into a standalone module runnable on more performance-conscious platforms. A stable release with periodic upgrade is accessible to the public at http://134.74.112.41/DQ. The system utilizes webform and produces the best execution plan based on the greedy hill-climbing method and/or dynamic programming method, in response to user specified or generated \(n\)-way join of \(n\) relations (presently \(n \leq 6\)) that are placed in distinct nodes of a distance defined network.

Figure 4 shows a snapshot of the output produced using a summary option. The summary output consists of a graphic presentation of the best-identified join sequence as well as the tabular form of operation breakdown and data transmission cost. The optimality criterion takes account of the minimization of the total amount of data exchange and transfer over the network. Various options can be selected to specify different executions such as the choice of whole-join or semijoin and the choice to compare overhead against a uniform cost network.

B. Experimental Results

We carried out simulation experiments to investigate the performance characteristics of the two optimization methods and the effects of network distances. The experiments were based on random generations of network distances, relations, and common columns for natural joins of distinct relations. Various measures were obtained for each of 3-way, 4-way, and 5-way joins for 500 rounds of distributed schedules. Table I shows a set of uniform distributions utilized to characterize simulation environments. Also Table II shows the scale of data transmission of each experiment.

<table>
<thead>
<tr>
<th>Table I</th>
<th>RANDOMIZED SIMULATION ENVIRONMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) network distance</td>
<td>minimum</td>
</tr>
<tr>
<td>(2) tuple count</td>
<td>10,000</td>
</tr>
<tr>
<td>(3) number of joined columns</td>
<td>1</td>
</tr>
<tr>
<td>(4) size of joined columns</td>
<td>4</td>
</tr>
<tr>
<td>(5) column count</td>
<td>50</td>
</tr>
<tr>
<td>(6) tuple size</td>
<td>(total of (2)) + 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table II</th>
<th>AVERAGE DATA TRANSMISSION PER EXPERIMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>av. distance</td>
<td>greedy method</td>
</tr>
<tr>
<td>3-way joins</td>
<td>5.58</td>
</tr>
<tr>
<td>4-way joins</td>
<td>6.51</td>
</tr>
<tr>
<td>5-way joins</td>
<td>6.55</td>
</tr>
</tbody>
</table>

Recall that if distance is not factored into the computation of cost then it would be possible for that case to yield a plan with higher cost than what is produced when distance is taken into account. The experiment clearly indicates the significant impact of network distance on join schedules. The execution
plan produced by ignoring network cost (in which all distances are equal to 1) is likely inferior to the execution plan in which distance is taken into account.

The plot in Figure 5’s left side shows the importance of network distance factor. The plot shows that the possibility of inferior schedules increases as the number of joins increases. The dynamic programming method (denoted by DP in the plot) shows that schedules inferior to the distance-incorporated case are highly likely to occur, i.e., 62% of 4-way join schedules and 86% of 5-way join schedules. Moreover, the greedy method rarely produces plans in distance-neglected cases that outperform the distance-incorporated case. The experiment shows that such cases remain at most 5% of the entire experiment. Figure 5’s right side indicates the loss caused by this limitation reaching up to 1/3 of the average transfer needed for optimal join processing. The greedy method reports this average error slightly lower; however as noted below, it often misses the optimal plan in both the uniform-cost and the distance-incorporated case.

As illustrated in Figure 6, the greedy method finds suboptimal solutions increasingly often as the number of joins increases. The plot in Figure 6’s left side shows the rates of suboptimal solutions, 3-way 12%, 4-way 57%, and 5-way 71% of 500 schedules. The growth rate is significant. The plot in Figure 6’s right side shows the average communication overhead caused by the suboptimal schedules. On average, it amounts to about 1/3 (28%~38%) of the average transfer needed for the join processing. Such a growth of inferior schedules provides clear evidence of the trade-off between accuracy and speed, and the growth is expected to continue for higher-degree joins. Therefore, some heuristics need to be introduced.

The experiment also confirmed the result of past work suggesting that semijoins should be utilized whenever applicable. The plot in Figure 7’s left side shows the growth of semijoin opportunities, i.e., 53% of 3-way joins, 69% of 4-way joins, and 82% of 5-way joins. Note that our implementation of join optimization compares the cost of whole-join and semijoin and takes the smaller one. Therefore, there is no data recorded in the experiment in which semijoin use incurs a higher cost solution. Figure 7’s right side shows the significance of the average gain made by the use of semijoins. Use of semijoins saves more than half (62%~67%) of the data-transfer cost required for the (optimal) processing of the join.

Unfortunately, there is a practical limitation on the use of dynamic programming. This approach is feasible for at most 5-way joins in real-time while the greedy method is acceptable up to 8-way joins. Various code improvements do not help to resolve the combinatorial explosion.

VI. Conclusions and Future Work

The purpose of these experiments was to quantify network savings of distributed query performance when optimization takes account of network topology, primarily inter-node distance. Our results show that incorporating these distances into the cost model usually produces significantly better plans in both of the evaluated search algorithms, greedy and dynamic programming. Also of note are the significant savings when semijoins are used. But again, these savings are even more significant when the cost model includes network distances.
Since the 1990’s observers have generally agreed that new database management systems architectures call for development of more sophisticated optimization techniques [7], [11], [12]. As noted earlier, n-way joins for large n are beyond the capabilities of the algorithm described here. Several methods have been proposed for overcoming this limitation. These include iterative dynamic programming, which combines the use of dynamic programming with heuristic pruning when the search space becomes too large [11]. Randomized algorithms like simulated annealing [10] have also been proposed. These are inherently based on hill climbing methods, with randomization to attempt transitioning away from local minima. Adaptive Query Processing is a newer development that further explores and/or modifies the search plan space during execution based on runtime feedback, like the actual cardinality of intermediate relations [8]. In light of our preliminary results, it is likely that inserting an improved cost model that incorporates inter-site distances could only increase the quality of plans produced by these newer optimization techniques.

The fact that distances are determined without additional communication overhead in a hypercube network makes this topology highly desirable. If the search space is reduced using logical tables, a hypercube also offers the advantage of being able to provide summary statistics for the aggregate relation.

Further research is needed to confirm that performance using aggregate statistics on logical tables offers execution performance comparable to the case of considering all separate sites and optimizing in the larger search space, but with more precise cost estimations. Moreover, it remains to be shown that incorporating network distances into cost estimation improves performance of newer optimization algorithms suitable for large DDFDs. Importantly, while theoretical results (see [14]) suggest a hypercube topology can be maintained with comparable overhead to other ad-hoc networks, the engineered hypercube might not confer great advantage under all conditions. If the overhead of maintaining a hypercube network actually approached the amount of data transmission saved in query optimization, its benefit would correspondingly reduced. The way in which data is distributed over the nodes in a network could also affect the relative advantage of the hypercube. Additional work is needed to determine the precise conditions of data distribution favorable for use of the hypercube as an engineered DDFD.

In future work we should also like to draw our simulation parameters from a more realistic setting than the random distributions in our experiments. Ideally, queries and data sources used in actual operations by specialized entities will be used to generate parameters. This would allow the use of hypercube embedded DDFDs to be validated for particular problem domains before deployment which would be particularly helpful in military environments.

VII. ACKNOWLEDGMENT

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES


