Optimizing Equijoin Queries In Distributed Databases Where Relations Are Hash Partitioned

DENNIS SHASHA and TSONG-LI WANG
New York University

Consider the class of distributed database systems consisting of a set of nodes connected by a high bandwidth network. Each node consists of a processor, a random access memory, and a slower but much larger memory such as a disk. There is no shared memory among the nodes. The data are horizontally partitioned often using a hash function. Such a description characterizes many parallel or distributed database systems that have recently been proposed, both commercial and academic. We study the optimization problem that arises when the query processor must repartition the relations and intermediate results participating in a multijoin query. Using estimates of the sizes of intermediate relations, we show (1) optimum solutions for closed chain queries; (2) the NP-completeness of the optimization problem for star, tree, and general graph queries; and (3) effective heuristics for these hard cases.

Our general approach and many of our results extend to other attribute partitioning schemes, for example, sort-partitioning on attributes, and to partitioned object databases.

Categories and Subject Descriptors: F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; H.2.4 [**Database Management**]: Systems—*distributed systems; query processing*; H.2.6 [**Database Management**]: Database Machines

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Equijoin, hashing, NP-complete problems, relational data models, spanning trees, systems

1. INTRODUCTION

We consider the class of distributed database systems consisting of a collection of nodes (sites), each having a processor, local memory, and local disk. The nodes communicate with one another across a high bandwidth (and high latency) network. Each relation is horizontally partitioned to different nodes

This work was partially supported by the National Science Foundation under grants DCR8501611 and IRI-8901699, and by the Office of Naval Research under grant N00014-85-K-0046.
Authors' addresses: D. Shasha, Courant Institute of Mathematical Sciences, New York University, New York, NY 10012; J. T. L. Wang, Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, New Jersey, 07102

Node 1:

R_1	Emp #	Salary
	2	55,000
	4	24,000

S_1	Emp #	Name	Age
	2	John	35
	4	Mary	25

Node 2:

R_2	Emp #	Salary
	1	38,000
	3	42,000

S_2	Emp #	Name	Age
	1	Andy	50
	3	Ann	40

Fig. 1. Example of two partitioned relations.

by hashing: use a hash function h whose domain is the domain of some attribute A of relation R and whose range is the number of processing nodes. For each tuple t of R, we put t in processing node $h(t.A)$.[1] When performing joins, we use the same function for any pair of relations that could possibly be joined together.

Example 1. Suppose we have a system composed of two nodes, node 1 and node 2, and a hash function $h(x) = (x \bmod 2) + 1$. Figure 1 shows two relations that are partitioned on the join attribute emp#: $R(S)$ is partitioned into R_1 and R_2 (S_1 and S_2), which are stored at node 1 and node 2, respectively.

Such a loosely coupled hash-partitioned architecture characterizes many parallel or distributed database systems that have recently been developed, including commercially available ones such as the Teradata machine [33], as well as research prototypes such as Bubba [9], Gamma [11], Grace [18], and Sabre [34].

In this paper we are concerned with optimizing equijoin queries in such systems. We restrict ourselves to queries that retrieve all fields of joined relations.[2] For our purposes, we assume that relations are not replicated, that is, each node holds a unique fragment for each relation. We also assume that there is no answer site in the systems. Thus we mainly concentrate on join processing, ignoring unioning results at the answer site.

For example, consider doing the join between R and S in Figure 1, based on the condition $R.\text{emp\#} = S.\text{emp\#}$. Since both relations are partitioned on emp#, no communication is required. On the other hand, if R is partitioned on emp#, whereas S on age, each S tuple t needs to be sent to processing

[1]Though one may partition on a set of attributes, for the purpose of this paper, we assume partitioning on a single attribute only. Some partitioning schemes (see, e.g., [18]) also allow the range to be k times the number of processing nodes, with each node having k buffers that can receive data. In that case, $h(t.A)$ designates a buffer as well as a node. The results in this paper hold for that generalization.

[2]This restriction is made mainly for simplifying cost computation. It is not difficult, however, to extend our techniques to handle queries whose outputs contain only certain fields of joined relations

Table I. Effect of Repartitioning on Join Performance

Join Size	With Repartitioning (seconds)	Without (seconds)
10,000 with 1,000	35.4	18 0
100,000 with 10,000	286 4	122.0
1,000,000 with 100,000	3,144.0	1,143.0

Source: DeWitt et al. [12].

node $h(t.\text{emp}\#)$. Such data transmission is called *repartitioning*. The overhead caused by repartitioning is not negligible. If communication among nodes is slow, then most time will be spent in routing data in a network when repartitionings occur. Even if communication is fast, repartitioning may reduce throughput because it entails more disk accesses.

To understand the importance of such overhead, let us review an experiment by DeWitt et al. on the Teradata machine [12]. The machine consisted of 20 nodes and 40 disk storage units, which were interconnected with a 12-Mbyte/second-bandwidth network. The data consisted of 200 byte tuples. DeWitt et al. performed joins between two relations having different sizes (10,000 tuples with 1,000 tuples, 100,000 tuples with 10,000 tuples, 1,000,000 tuples with 100,000 tuples, respectively) and recorded the time for executing these joins with and without repartitionings. Table I shows the result of their experiment. By comparing the values listed in the second and third columns of the table, we see that there exist ratios of between 2 or 3 to 1 for the same size joins. This indicates that when processing queries in a loosely coupled multiprocessor system, the repartitioning cost can dominate the local processing cost. The authors also measured the influence of the bandwidth of the network. They observed that repartitioning a relation of 200 Mbyte took 2,000 seconds and used less than 1 percent of network capacity on the average. This result reveals that repartitioning large relations is expensive even when a network is fast.

But can we control the repartitionings necessary to process a query, thereby minimizing response time in such an environment?

Example 2. Suppose we have a suppliers-and-parts database containing relations SUP(s#, sname, city), PART(p#, pname, city), and SUP-PART(s#, p#, qty). Suppose that SUP is partitioned on s#, PART on p#, and SUP-PART on s#. Consider the following SQL query:

```
SELECT  SUP.*, PART.*, SUP-PART.*
FROM    SUP, PART, SUP-PART
WHERE   SUP.s# = SUP-PART.s#
AND     SUP-PART.p# = PART.p#
```

One way to process this query is to join SUP-PART with PART first, obtaining a result T_1, and then join T_1 with SUP. Assume that joining two relations takes 100 seconds and repartitioning a relation takes 100 seconds, as indicated in Table I. The cost of this strategy would be (the cost of repartitioning SUP-PART) + (the cost of joining SUP-PART with PART) + (the cost of

repartitioning T_1) + (the cost of joining T_1 with SUP) = 100 + 100 + 100 + 100 = 400 seconds. On the other hand, joining SUP with SUP-PART first requires no repartitioning. Then joining the intermediate result T_2 with PART requires repartitioning T_2. The cost of this strategy is thus 100 + 100 + 100 = 300 seconds.

Clearly, the second strategy is better for processing the given query. This example demonstrates that by properly arranging the order of joins among relations, better performance can be achieved. Exactly how to do this to minimize the cost for various important queries is the main concern of this paper. In Section 2, we present an overview of previous work in the area. Section 3 discusses basic assumptions and formally defines the optimization problem. Section 4 then gives a dynamic programming algorithm to solve the problem in the context of closed chain queries. Section 5 establishes the NP-completeness of the problem for star, tree, and general graph queries. Section 6 presents and analyzes heuristic procedures. In Section 7, we report the results of computational experiments. Finally we conclude the paper with some directions for future research in Section 8.

2. RELATED WORK

A considerable amount of work has been performed in the area of query processing using partitioning schemes. Bernstein et al. [3], Stonebraker and Neuhold [31], and Williams et al. [35] discussed horizontal partitioning (fragmentation) in the context of distributed database management systems (DDBMSs). In these systems, fragmentation is used as part of processing strategy to increase parallelism, thus increasing throughput and saving response time.

A common objective adopted by researchers in the area of DDBMS is to minimize the communication cost incurred in processing a query. Segev [25], for example, introduced the notion of remote semijoin, which can significantly reduce the communication cost while incurring higher processing cost. Fragments may also be replicated at different sites. Yu et al. [38] distinguished queries as locally processable and nonlocally processable. If a query is locally processable, then the query can be processed without data transfer. They proposed a linear time "fragment and replicate" algorithm for nonlocally processable queries. Relevant work has also been discussed by Apers et al. [1], Wong [36], Pramanik and Vineyard [22], Stamos and Young [30], and others. What distinguishes us from these workers is that we seek the optimal join order when processing queries, starting with partitioned data and ending with partitioned data. Moreover, our main concern is on the environment without data replication. We defer to Section 8 the possibility of replicating data.

The idea of using hash partitioning for joins in multiprocessor systems comes from [2]. This scheme is then adopted by Kitsuregawa et al. on the Grace database machine project [18], though their emphasis is on the speed of the sort engine rather than on the performance of the join algorithm. Valduriez and Gardarin [34] described also various join and semijoin strate-

gies, but hashing is applied only during the partition phase. DeWitt and Gerber [10] proposed two algorithms that exploit hashing thoroughly in both partition and join phases. They applied the algorithms to a number of simple join operations and concluded that both algorithms have satisfactory performance.

In contrast to the previous work on hash partitioning, this paper examines *multijoin* queries arising in systems that use hash methods to execute joins and partition data. Thus, we are not concerned with the particular hash-join strategy used, but rather with reducing the cost of a query involving many joins by minimizing communication, I/O, and processing time. In [28], we developed a model for processing such queries without considering the size of relations and intermediate results participating in a join. Here, we address more general cases and explore strategies for taking size into account. We present a polynomial time algorithm for optimizing queries whose closure is a chain. We then show that finding optimal solutions to a simpler problem is NP-complete for star and tree queries. In view of the NP-hardness for the simpler problem, we consider various heuristics for the general problem. We find that combining our chain algorithm with a heuristic related to Kruskal's spanning tree algorithm [20] achieves the best performance over a wide range of query topologies and cost assumptions.

3. FORMULATION OF THE PROBLEM

3.1 Basic Assumptions

Our objective is to minimize response time when processing a query. We approximate this goal by considering both processing and repartitioning costs. Processing cost refers to the time spent for performing joins at each node. Repartitioning cost refers to the time spent for repartitionings, which is determined by (1) the total amount of data transmitted in the network, and (2) system parameters of a given architecture. System parameters include the bandwidth of the network, as well as the speed of the I/O and processors.

To simplify the model developed in this paper, we make the following assumptions.

Assumption 1. The processing speeds and I/O speeds at different nodes are the same. The bandwidth of the network is a constant.

Assumption 2. Each processing node uses the same hashing technique to execute joins (e.g., DeWitt and Gerber's).

Assumption 3. For each repartition (or partition) of a relation, the hash function applied to its attribute evenly distributes tuples to each node.

Assumptions 1 and 2 deal with parameters related to a given system. The two assumptions, together with Assumption 3, ensure that the processors complete joins and transfer data at the same time. (The data could be transferred either in pipelined fashion or one batch at a time.) Assumption 3 is reasonable when the attribute is a key and the hash function is "good," that is, it acts as a random function from the set of tuples to the set of n

nodes such that each tuple has probability $1/n$ of being sent to any node. This has been established analytically in [29] where we show that the probability that any of the n nodes has more than twice the average number of tuples is very small, provided that the number of tuples $> 4n \log n$. If the attribute is not a key or the hash function is biased, our results should be taken as heuristics.

3.2 Terminology

We assume the reader is familiar with the standard terms, attribute, tuple, join, used in relational database systems. Define a *relation schema* **R** as a finite set of attributes $\{A_1, A_2, \ldots, A_n\}$ [16]. Associated with each attribute A_i, $1 \leq i \leq n$, is a domain, denoted $\text{dom}(A_i)$. A relation instance (or simply *relation*) R on schema **R** is a finite set of mappings $\{t_1, t_2, \ldots, t_m\}$ from **R** to the set of domains such that for each $t \in R$, $t(A_i) \in \text{dom}(A_i)$, $1 \leq i \leq n$. Let B be an attribute in **R**. We write $t.B$ for $t(B)$. We define $R.B$ to be $\{t_1.B, t_2.B, \ldots, t_m.B\}$.

An equijoin 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 $R.B = S.C$. We are only interested in queries whose qualification is a conjunction of such join conditions. Therefore, we define a *query* as a set of clauses.[3] To represent a join that links different tuples of the same relation, we consider the two arguments of the join to be different relation instances for the purpose of the query. For example, to find the salary of each employee's current manager, given an EMP(name, sal, manager) schema, we assume two employee relation instances EMP1 and EMP2. These would be represented as distinct relations in the query.

In order to find an optimal join strategy, it is helpful to consider the *query graph* $QG_q = (QV_q, QE_q)$ for a query q, where

$$QV_q = \{\text{relations referenced by the clauses in } q\}$$

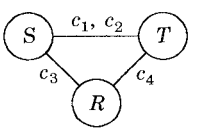
$$QE_q = \{\{R, S\} \mid \text{some member of } q \text{ references both } R \text{ and } S\}.$$

An example of a query graph is given in Figure 2a. Notice that a 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 equijoin clauses referencing both R and S in q. We assume 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 these individual queries.

Observe that in processing a query graph, one can always execute a spanning tree of it, treating all other clauses as selections. Furthermore, in executing a spanning tree, we can take only one clause, for each edge, as the

[3] Because all attributes of joined relations appear in the output of a query, as marked in Section 1, we exclude the target list in defining a query.

SELECT	$S.*, T.*, R.*$
FROM	S, T, R
WHERE	$S.C = T.B$
AND	$S.G = T.H$
AND	$S.F = R\ E$
AND	$T.B = R.D$

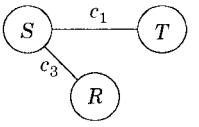


$c_1 = \{S.C, T.B\}$
$c_2 = \{S.G, T.H\}$
$c_3 = \{S.F, R.E\}$
$c_4 = \{T.B, R.D\}$

(a)

(1) Consider the query graph in Figure 2a just above. We can execute the graph by joining S with T first, and then ST with R. (The juxtaposition of two relation names represents the join result of the relations.)
(2) While joining S and T based on, say $\{S.C, T.B\}$, the selection condition $S.G = T.H$ is also processed.
(3) Then when joining ST with R based on, say $\{S.F, R.E\}$, the clause $\{T.B, R.D\}$ enforces another selection condition on tuples of ST and R (see Figure 2c).

(b)



(c)

(4) Consider the query graph in Figure 2a. When joining S and T on the clause $\{S.C, T.B\}$, S needs to be partitioned on C and T on B. Then in joining ST and R,
  (i) if $\{S.F, R.E\}$ is used as the join clause, ST needs to be repartitioned on F; such a repartitioning causes part(T) to be empty;
  (ii) if $\{T.B, R.D.\}$ is used as the join clause, since no repartitioning of ST is needed, part(T) = $\{B\}$.

Fig. 2. Example of query graph and its single-clause spanning tree.

join clause, applying the other clauses as selections (see Figure 2b). This motivates us to introduce the following.

Definition 1. A *spanning tree* of a query graph $QG_q = (QV_q, QE_q)$ is a graph $QG_{Treeq} = (QV_{Treeq}, QE_{Treeq})$ with the following properties:

(i) $QV_{Treeq} = QV_q$.
(ii) $QE_{Treeq} \subseteq QE_q$ and (QV_{Treeq}, QE_{Treeq}) is a tree.
(iii) For each $e \in QE_{Treeq}$, clauses(e, *Treeq*) $\subseteq$ clauses(e, q) and clauses(e, *Treeq*) $\neq \emptyset$.

If clauses(e, *Treeq*) is a singleton set for each $e \in QE_{Treeq}$, QG_{Treeq} is called a single-clause spanning tree (ss tree) of QG_q.

Figure 2c shows an ss tree for the query graph in Figure 2a, which corresponds to the execution sequence in Figure 2b.

3.3 Cost Model

We denote as part(R) the set of attributes on which a relation R is partitioned sometime in the query execution. By the assumption stated in Section

1, the initial value of part(R) is always a singleton set. However, when repartitionings occur, part(R) may become empty. Figure 2d illustrates such a case.

In order to evaluate a join strategy effectively, we need to have a measure that reflects the cost. We begin with the definition of the cost for a clause.

Definition 2. Let $|R|$ denote the number of tuples in relation R and $|t_R|$ the width (in bytes) of a tuple in R. The cost of a clause $\{R.A, S.B\}$ equals PC + RC; PC = $\alpha \times (|R| \times |t_R| + |S| \times |t_S|)$ is the processing cost, and RC = $\beta \times Move$ is the repartitioning cost, where α, β are nonzero constants (e.g., in the Teradata case $\beta = 2\alpha$), and

$$Move = \begin{cases} 0 & \text{if } (A \in \text{part}(R)) \wedge (B \in \text{part}(S)) \\ |S| \times |t_S| & \text{if } (A \in \text{part}(R)) \wedge (B \notin \text{part}(S)) \\ |R| \times |t_R| & \text{if } (A \notin \text{part}(R)) \wedge (B \in \text{part}(S)) \\ |R| \times |t_R| + |S| \times |t_S| & \text{otherwise.} \end{cases}$$

Thus we are assuming that both PC and RC are linearly proportional to the sizes of the relations. This assumption is consistent with the Teradata benchmarks shown in Section 1 and the Grace measurements [18, 19]; it is also consistent with the cost equations derived elsewhere [27]. *Move* represents the amount of data transmitted in the network, where case 1 refers to the situation in which neither R nor S will be repartitioned; cases 2 and 3 represent the situation where one of the relations will be repartitioned, and case 4 represents the situation where both relations need to be repartitioned.

Also, we consider the measure of processing cost to be the number of bytes moved from disks, as opposed to the number of page fetches from disks often assumed in the literature [4, 17, 24], though the two measures differ by only a constant additive factor. By including factors such as page size, I/O speed, and bandwidth, one can convert the byte units to corresponding time unit.

Next, as noted in Section 1, in executing the same ss tree of a graph, joining nodes in different order can yield different costs.[4] We thus introduce the following.

Definition 3. A clause string, or *string* for short, associated with an ss tree $QG_{Treeq} = (QV_{Treeq}, QE_{Treeq})$ is an *ordered* clause sequence

$$c_{i1}\, c_{i2} \ldots c_{in}$$

where $QE_{Treeq} = \{e_i \mid 1 \le i \le n\}$, $i1, i2, \ldots, in$ is a permutation of $1, 2, \ldots, n$, and $c_{ij} \in \text{clauses}(e_{ij}, Treeq)$.

The order of clauses in a string indicates the order in which nodes of the associated tree are joined. There are $n!$ strings associated with QG_{Treeq} if there are n edges in it.

[4]Since the results obtained in this paper depend on the properties of graphs, we use the terms node and relation interchangeably.

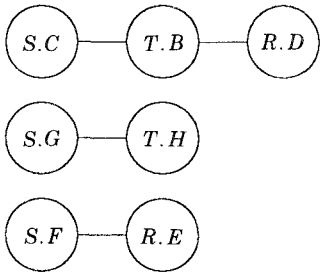


Fig. 3. Join graph of query shown in Figure 2a.

We define the cost of a string as the sum of the costs of its component clauses. However, since the cost of a clause is dependent on the sizes and partition attributes of its two corresponding relations, which in turn are influenced by the order in which relations of the query tree are joined, it is required that, when calculating the cost of a string, one follow the exact order specified in the string.

Finally, we define the cost of an ss tree QG_{Treeq} as

$$\text{Cost}(QG_{Treeq}) = \min_{cs \in CS} \{\text{Cost}(cs)\}$$

where CS is the set of all strings associated with QG_{Treeq}. A *minimum cost spanning tree* of a query graph QG_q is an ss tree of QG_q such that no other ss tree of QG_q has a smaller cost. For a given query graph, executing its minimum cost spanning tree yields minimum response time.

3.4 Exploiting Redundant Clauses to Optimize Queries

Our goal is to minimize response time when processing a query. Thus far we have concentrated on query graphs, rather than on queries. However, different query graphs may refer to the same query. To have a clear view of this, we need another graph model.

Define the *join graph* for a query q, denoted JG_q, as a pair (JV_q, JE_q), where

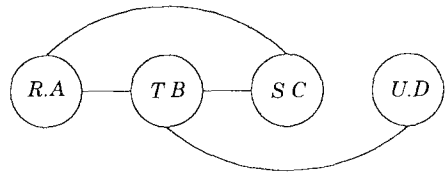
$$JV_q = \{R.A \mid R.A \text{ is an attribute of } R \text{ that occurs in some clause of } q\}$$

$$JE_q = \{\{R.A, S.B\} \mid \{R.A, S.B\} \text{ is some clause in } q\}.$$

Intuitively a join graph represents an equivalence relation on attributes from different relations. (This is because we consider only queries with equijoin clauses.) Each equivalence class of attributes is a connected component of the graph. Figure 3 shows the join graph of the query in Figure 2a.

We say two queries are *equivalent* if their join graphs have the same connectivity. For any state of a database, equivalent queries give the same result. A clause c of a query q is *redundant* if c is an edge in a cycle of JG_q. The *closure* of a query q is an equivalent query, denoted q^+, to which one cannot add more redundant clauses. Figure 4 illustrates the above concepts. A query is a *chain query* (*star query*, *tree query*, respectively) if it is equivalent to some query whose query graph is a chain (star, tree,

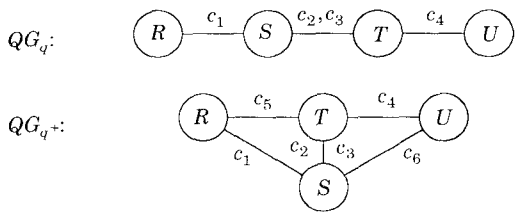
(1) Consider the query $q = \{\{R.A,T\ B\}, \{T\ B,S\ C\}, \{R.A,S\ C\}, \{T.B,U.D\}\}$.
(2) q's join graph



(3) $q' = \{\{R\ A,T.B\}, \{T\ B,S\ C\}, \{T.B,U\ D\}\}$ is equivalent to q.
(4) $\{R.A,S.C\}$ is redundant.
(5) $q^+ = \{\{R.A,T.B\}, \{T.B,S.C\}, \{R\ A,S\ C\}, \{T.B,U.D\}, \{R\ A,U.D\}, \{S\ C,U.D\}\}$.

Fig. 4. Join graph representation of a query and its closure

(1) Consider the query q and its closure (expressed in terms of their query graphs):



$c_1 = \{R.A,S.B\}$
$c_2 = \{S\ B,T\ D\}$
$c_3 = \{S\ C,T\ E\}$
$c_4 = \{T.E,U\ F\}$
$c_5 = \{R\ A,T.D\}$
$c_6 = \{S.C,U.F\}$

(2) Assume that all relations and intermediate results have the same size, K. Moreover, part(R) = A, part(S) = C, part(T) = D, and part(U) = F
(3) If we only consider QG_q, we get the best strings (e.g., $\{R.A,S\ B\}\{S.B,T.D\}\{T\ E,U.F\}$) with cost 10K.
(4) If we consider the redundant clauses in QG_{q^+}, we can get the string $\{R\ A,T.D\}\{T.E,S.C\}\{T.E,U\ F\}$ with cost 8K.

Fig. 5. Example showing the need to compute the query closure.

respectively). The importance of these queries and their processing in distributed database systems have been discussed extensively in the literature [5–7, 16, 37].

In order to minimize costs when processing a query, one might conjecture that it is enough to look at the spanning trees of its query graph. But this is wrong. Figure 5 illustrates such a case.[5] This example highlights an important idea: redundant clauses may lead to cheaper strings (and ss trees). It also suggests that when processing a query, we should begin with the closure of the query.

[5]For concreteness, in this and subsequent examples, we set the constants α and β in the cost model (cf. Definition 2) to 1 and 2, respectively (two values that are consistent with the Teradata benchmark result)

Computing closures may sometimes introduce *self-loops* (i.e., edges from nodes to themselves) in the query graph. For example, suppose a given query q has clauses $\{R.A, S.B\}$, $\{S.B, T.C\}$, and $\{T.C, R.D\}$. Then QG_{q+} will contain $\{R.A, R.D\}$, an edge from R to R. In general, one can eliminate a loop $\{R.X, R.Y\}$ by replacing either $R.X$ by $R.Y$ or $R.Y$ by $R.X$ in all clauses of the query. The choice is arbitrary unless one of them, say $R.Y$, is the initial partition attribute of R, in which case we replace $R.X$ by $R.Y$. $\{R.X, R.Y\}$ will then be removed from the query graph and used as a selection condition. Eliminating self-loops is useful, since by renaming identical attributes we may reduce costs. Henceforth, we restrict our attention to loop-free query graphs.

Based on the above formulation, the problem of minimizing response time (MRP) becomes

> **Given** a query q,
> **Find** (1) QG_{q+}; (2) a minimum cost spanning tree $\mathcal{T}$ of QG_{q+} and the clause string cs that yields $\mathcal{T}$'s cost.

In the following sections, the problem is addressed for chain, star, tree, and general graph queries. It is important to note that when processing these queries, once the spanning tree $\mathcal{T}$ and associated string cs are found, we can process them by executing $\mathcal{T}$ based on the clauses and join order specified in cs.

4. CLOSED CHAIN QUERIES

As shown in Figure 5, the closure of a chain in the query graph does not necessarily remain a chain. If it is not a chain, it will contain a cycle.[6] In this section, we focus on queries q whose QG_{q+} is a chain. (Such a class of queries are called *closed chain queries*.) More general queries are treated in the next section.

A dynamic programming technique is used to solve the MRP problem for closed chain queries. The dynamic programming algorithm computes the best order of joins. The algorithm, in its style, is similar to that proposed independently by Sun et al. [32] for optimizing chain queries in object-oriented database systems.

The following notation is needed to describe our algorithm. For $1 \le i \le j \le n$,

(1) $cs(i, j)$: a minimum cost string associated with the chain segment $R_i, R_{i+1}, \ldots, R_j$;
(2) Cost(i, j): the cost of $cs(i, j)$;
(3) R_{ij}: the intermediate relation obtained by joining the relations $R_i, R_{i+1}, \ldots, R_j$ based on the clauses and join order specified in $cs(i, j)$.

In general, when joining any two (possibly intermediate) relations, say R_{ip} and R_{p+1j} for some p, $i \le p < j$, among the clauses between the two

[6] To see this, notice that connecting two nonadjacent nodes on a single path results in a node becoming a descendant of the two nodes.

relations, we want to choose a pair with the lowest cost as the join clause. Observe that the processing cost depends solely on the sum of the sizes of the two relations, and hence is the same for all these clauses. We thus choose an attribute, if any, on which no repartitioning is needed, thereby minimizing the communication cost.

After joining the two relations on a clause $\{R_p.X, R_{p+1}.Y\}$, the cardinality of the new intermediate relation R_{ij} is $|R_{ip}| \times |R_{p+1j}| \times \alpha$, where α is the selectivity factor between R_{ip} and R_{p+1j},[7] and the width of tuples in R_{ij} is $|t_{Rip}| + |t_{Rp+1j}|$.[8] Algorithm CHAIN summarizes the procedure.

Algorithm CHAIN

1. (*Initialization*) For $1 \le i \le n$, $cs(i, i)$ = null, $\text{Cost}(i, i) = 0$, $R_{ii} = R_i$, part(R) is the attribute that R is initially partitioned on;
2. **for** $k = 1$ **to** $n - 1$ **do**
3. **for** $i = 1$ to $n - k$ **do**
4. $j = i + k$;
5. **for** $p = i$ **to** $j - 1$ **do**
6. choose clause $\{R_p.X, R_{p+1}.Y\}$ such that
$\text{Cost}(\{R_p.X, R_{p+1}.Y\}) = \min_{c \in \text{clauses}(\{R_p, R_{p+1}\}, q)}\{\text{Cost}(c)\}$;
7. concatenate $cs(i, p)$, $cs(p + 1, j)$, $\{R_p.X, R_{p+1}.Y\}$ to form a string cs;
set $\text{Cost}(cs) = \text{Cost}(i, p) + \text{Cost}(p + 1, j) + \text{Cost}(\{R_p.X, R_{p+1}.Y\})$;
put cs in set(i, j);
end for
8. construct $cs(i, j)$ where $\text{Cost}(i, j) = \min_{cs \in set(i,j)}\{\text{Cost}(cs)\}$;
compute the size of R_{ij} as described above;
end for
end for

It should be noted that $cs(1, n)$ is not unique if all the clauses (strings) with the same minimum cost are selected at steps 6 and 8.

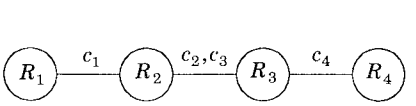
To show that the algorithm solves the MRP problem for closed chain queries, it suffices to prove the following.

THEOREM 1. *Given a chain segment with relations $R_i, R_{i+1}, \ldots, R_j$, algorithm CHAIN finds a minimum cost string of the segment.*

PROOF. The proof is by induction on $j - i$, the length of the segment. The base case ($j - i = 1$) is straightforward. Assume the theorem holds for any chain with length less than or equal to $k - 1$. Let $\tilde{cs}(i, j)$ be a string of the segment with the actual minimum cost, where $j - i = k$. It will be proved that $\text{Cost}(\tilde{cs}(i, j)) = \text{Cost}(i, j)$. Note that $\tilde{cs}(i, j)$ is not necessarily equal to

[7] It may happen that, when joining two (intermediate) relations, there exist clauses whose selectivities differ from that of the chosen join clause. As an example, consider the optimal string $\{R.A, T.D\}\{T.E, S.C\}\{T.E, U.F\}$ in Figure 5. When joining RT with S, the selectivity of the clause $\{R.A, S.B\}$ may differ from that of $\{T.E, S.C\}$. For simplicity, we define the selectivity factor between two (intermediate) relations R and S as the fraction of tuple pairs from R and S satisfying *all* clauses between them. Note that, under this definition, the cardinality of a join result is independent of which clause is used as the join clause.

[8] Strictly speaking, the width should be $|t_{Rip}| + |t_{Rp+1j}| - |X|$—it is sufficient to retain one join attribute in the resulting relation because equijoins are involved. For exposition purpose, we omit the last term and include it only in the simulation study (Section 7).



$c_1 = \{R_1.A, R_2.B\}$
$c_2 = \{R_2.C, R_3\ D\}$
$c_3 = \{R_2.E, R_3.F\}$
$c_4 = \{R_3.G, R_4.H\}$

Fig. 6. Query graph for closed chain query example.

$cs(i, j)$, the minimum cost string obtained by algorithm CHAIN for this segment.

Assume that the last clause joined in $\tilde{cs}(i, j)$ is $\{R_p.U, R_{p+1}.V\}$. Let $\tilde{cs}(i, p)$ be the string obtained by concatenating the clauses from $\tilde{cs}(i, j)$ that are on the edges between R_i and R_p, and $\tilde{cs}(p + 1, j)$ be the string formed by concatenating the remaining clauses except $\{R_p.U, R_{p+1}.V\}$, with the constraint that the order of clauses in the two strings obeys that in $\tilde{cs}(i, j)$. By induction hypothesis, $\text{Cost}(\tilde{cs}(i, p)) \geq \text{Cost}(i, p)$ and $\text{Cost}(\tilde{cs}(p + 1, j)) \geq \text{Cost}(p + 1, j)$.

Now, $\text{Cost}(\tilde{cs}(i, j)) = \text{Cost}(\tilde{cs}(i, p)) + \text{Cost}(\tilde{cs}(p + 1, j)) + \text{Cost}(\{R_p.U, R_{p+1}.V\}) \geq \text{Cost}(i, p) + \text{Cost}(p + 1, j) + \text{Cost}(\{R_p.U, R_{p+1}.V\}) \geq \text{Cost}(i, j)$ (steps 6 and 8). Since $\text{Cost}(\tilde{cs}(i, j))$ is the known minimum, we have $\text{Cost}(\tilde{cs}(i, j)) = \text{Cost}(i, j)$. □

The following shows the time complexity of the algorithm.

THEOREM 2. *Algorithm CHAIN runs in $O(n^3 + n^2 l)$ time, or $O(n^3)$ when n is greater than l, where n is the number of nodes in a closed chain and l is the maximum number of clauses on an edge.*

PROOF. Consider the following two cases. When $k = 1$, we are joining two relations rather than intermediate results. Thus in order to find the lowest cost clause, we need to examine all clauses between the two relations, checking whether their component attributes are the same as the initial partition attributes of the two relations. This requires $O(nl)$ time.

When $k > 1$, in joining R_{ip} and R_{p+1j} for some p, $i \leq p < j$, we need to examine all clauses between them only when $i = p$ but $p + 1 < j$ (or $i < p$ but $p + 1 = j$), because in such situations we are joining a relation with an intermediate result, and we need to check whether any attribute matches the initial partition attribute of the relation. In the other situations, in which we are joining two intermediate relations, we can arbitrarily choose a clause, since a closed chain does not allow the same attribute to occur on distinct edges (because then the closure would not be a chain). Therefore, both intermediate relations need to be repartitioned anyway. Thus, this requires $O(n^2 + nl)$ time.

The value of k ranges from 1 to n. Hence, there are $O(n^3 + n^2 l)$ operations to be performed in total. □

Example 3. Consider the query graph in Figure 6. Both the initial data and results obtained by applying algorithm CHAIN to this graph are shown

Table II. Data for the Closed Chain Query Example

i	1	2	3	4
		Cardinality of Relation		
$\|R_i\|$	30	10	10	20
		Initial Partition Attribute		
part(R_i)	U	C	D	V
		Tuple Size		
$\|t_{R_i}\|$	3	2	1	4
		Selectivity Factor		

α_{12}[a]	α_{23}	α_{34}
0 2	0.1	0.2

Results

i	j	Cost(i, j)	$cs(i, j)$	$\|R_{ij}\|$	$\|t_{R_{ij}}\|$
1	2	330	c_1	60	5
2	3	30	c_2	10	3
3	4	270	c_4	40	5
1	3	390	c_2c_1	60	6
2	4	360	c_2c_4	40	7
1	4	1470	$c_2c_4c_1$	240	10

[a] α_{ij} is the selectivity factor between R_i and R_j.

in Table II. Thus the minimum cost required to process the query is 1470 and the corresponding string is $\{R_2.C, R_3.D\}\{R_3.G, R_4.H\}\{R_2.B, R_1.A\}$.

5. MORE GENERAL QUERIES

Solving the MRP problem for a complex query whose graph has multiple edges, with each edge being associated with multiple clauses, is NP-complete. The difficulty stems from simultaneous choices among multiple clauses for multiple edges. In this section, we address the time complexity of the MRP problem for such complex queries. In particular, we focus on star queries, a particularly simple form of tree queries.

THEOREM 3. *Given a star query q without redundant clauses, finding a minimum cost spanning tree from QG_q is NP-complete.*

PROOF. The proof that this problem can be solved by a nondeterministic algorithm in polynomial time is straightforward: given a target cost, just guess a single-clause spanning tree and see whether its cost is less than the target cost. To show that the problem is NP-hard, we consider the following easier case. The size of each relation in QG_q is the same, and the result of each join is as large as the relations participating in the join. Under these restrictions, minimizing response time is equivalent to minimizing the *number* of repartitionings.

We therefore modify the cost of an ss tree $QG_{Treeq} = (QV_{Treeq}, QE_{Treeq})$ to be

$$\text{Cost}(QG_{Treeq}) = \sum_{R \in QV_{Treeq}} \text{nodecost}(R)$$

where the nodecost of a relation is the number of nonpartitioning attributes in clauses touching that relation.

Intuitively this cost reflects the smallest number of repartitionings that the execution of an ss tree requires. To show this, consider some node R in QG_{Treeq}. Since QG_{Treeq} is a single-clause spanning tree, if $R.A$ appears in a clause, then R or an intermediate result containing R will have to be partitioned on A at some point in the query execution. If A is the partition attribute of R, then R may not have to be repartitioned to allow the joins associated with $R.A$ to occur. Thus, the nodecost of R 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 tree.[9] So the number of repartitionings must be at least the sum of these nodecosts.

We show that the special case is NP-hard by transforming the following hitting set problem [14] to it.

Hitting Set. Given a collection $\mathscr{C}$ of subsets of a finite set S and a positive integer K, decide whether there is a subset S' of S with $\|S'\| \leq K$ such that S' contains at least one element from each subset in $\mathscr{C}$.

We construct a query graph QG by the following steps:

(1) For each subset $C_i \in \mathscr{C}$, create a node R_i. Create an additional distinguished node R.

(2) If C_i contains members $B_{i1}, B_{i2}, \ldots, B_{ik_i}$, associate the edge $\{R, R_i\}$ with the clauses $\{R.B_{i1}, R_i.B_{i1}\}, \{R.B_{i2}, R_i.B_{i2}\}, \ldots, \{R.B_{ik_i}, R_i.B_{ik_i}\}$.

See Figure 7 for an example of the above construction, where $\|\mathscr{C}\| = m$. Assume that none of the B_{ij} are partition attributes. It is easy to verify that the resulting graph has the following properties.

(a) All clauses between R and each R_i are of the form $\{R.X, R_i.X\}$ for some attribute X.

(b) None of the clauses are redundant.

(c) Each R_i has a nodecost of one in any ss tree of QG.

Consider QG as the query graph of a certain star query q. Let us suppose that there is a single-clause spanning tree of QG_q with cost $\|\mathscr{C}\| + K$. It is then easily seen that there is a hitting set of $\mathscr{C}$ with K elements (by property (c) above and the fact that R has a nodecost of K). Conversely if $\mathscr{C}$ has a

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

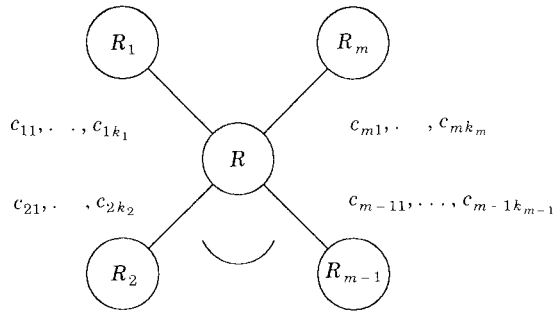


Fig. 7. Transforming $\mathscr{C} = \{C_1, C_2, \ldots, C_m\}$, where $C_i = \{B_{i1}, B_{i2}, \ldots, B_{ik_i}\}$, $1 \le i \le m$, into QG. R is a distinguished relation; each R_i corresponds to C_i; each clause c_{ij} corresponds to the join condition $R_i.B_{ij} = R.B_{ij}$, $1 \le j \le k_i$, that is, $c_{ij} = \{R_i.B_{ij}, R.B_{ij}\}$, $1 \le j \le k_i$, $1 \le i \le m$

hitting set H with K elements, we can construct a single-clause spanning tree of QG_q by associating one clause $\{R.X, R_i.X\}$ with each edge where $X \in H \cap C_i$. (Note that if H includes more than one element from a subset C_i, we arbitrarily choose one element as the attribute in the corresponding clause.) The cost of this tree is at most $\| \mathscr{C} \| + K$. Thus, the special case is NP-hard. It follows that our problem is also NP-hard. □

The above theorem says that it is already hard when only considering the query, rather than its closure. The next theorem shows that the same result holds for the query's closure.

THEOREM 4. *Problem (MRP) is NP-complete for star queries.*

PROOF. We adopt the cost formula in Theorem 3 and again transform the hitting set problem to the current problem; that is, we must show that there is an ss tree of QG_{q+} with cost $\| \mathscr{C} \| + K$ iff there is a hitting set with K elements.

(1) If. This follows from the proof in Theorem 3 since there would then be an ss tree of QG_q with cost $\| \mathscr{C} \| + K$. That tree would also be an ss tree of QG_{q+}.

(2) Only if. Let $\mathscr{T}$ be an ss tree of QG_{q+} with cost $\| \mathscr{C} \| + K$. We construct an ss tree of QG_q with the same or less cost. The proof in Theorem 3 then implies the existence of a hitting set of $\mathscr{C}$ with at most K elements.

Observe that the cost of any ss tree in QG_{q+} or QG_q is the sum of the number of distinct attributes touching each node. Let us make the distinguished relation R be the root of the given ss tree $\mathscr{T}$. Let the nodes of depth 1 in $\mathscr{T}$ be the neighbors of R. In general, the nodes of depth i, $i \ge 1$, in $\mathscr{T}$ will be those whose shortest simple path to R contains i edges. Now, for each attribute X, we reconnect edges by the following steps. As a result, certain nodes that were not neighbors of R will become R's neighbors.

For each clause $\{R_{k1}.X, R_{k2}.X\}$ that connects the depth $i+1$ node R_{k1} to the depth i ($i \geq 1$) node R_{k2} (we know that the two nodes share a common attribute, here X, by the construction of q and the definition of redundancy),

—eliminate that clause and its corresponding edge;
—create the edge $\{R_{k1}, R\}$, associating the edge with $\{R_{k1}.X, R.X\}$. (We know that q has $\{R_{k1}.X, R.X\}$ and $\{R_{k2}.X, R.X\}$, because q and qt are equivalent and $\{R_{k_1}.X, R_{k_2}.X\}$ could not be inferred from q otherwise.)

Two cases may arise.

Case 1. There is already a clause connecting to R using attribute X. So, the nodecost of R does not change, and the nodecost of R_{k_1} may or may not decrease, but certainly will not increase.

Case 2. There is no clause connecting to R using attribute X. Therefore, some edge in the path from R_{k_2} to R does not have X on it; say the edge from R_{k_3} to R_{k_4} is the first one. In that case, removing the edges leading to R_{k_3} that contain a clause with attribute X will reduce the nodecost of R_{k_3} by one. At most, the nodecost of R can increase by one, so the cost of the resulting tree cannot increase.

We perform such reconnections for each attribute in turn, so that each node other than R in the resulting tree is connected only to R by a single clause in q. Thus, the resulting tree is an ss tree of QG_q. Furthermore, since the reconnections for each attribute do not increase any cost, the cost of the ss tree will be at most $\| \mathscr{C} \| + K$. □

Because stars are subcases of trees and general graphs, we obtain the following immediately.

COROLLARY 1. *Problem (MRP) is NP-complete for tree queries.*

COROLLARY 2. *Problem (MRP) is NP-complete for general graph queries.*

The results obtained so far give us a picture for various queries shown in Figure 8.[10] These results are discouraging as they indicate that even for simple queries, an enumerative algorithm must be used to find the optimal join strategy. In the next section, we develop heuristics that approximate the optimal solution while having a polynomial time complexity.

6. HEURISTIC ALGORITHMS

Our general approach to processing a complex query is as follows. Given a query q, we first construct the query graph of its closure, QG_{q+}. We then

[10] *Closed tree queries* in the figure are defined in a similar way as closed chain queries; namely, they refer to tree queries whose closures are trees. As the reader may have noticed, the construction of QG_q in Theorem 3 implies that QG_{q+} may contain a cycle. This then raises an interesting question as to whether there exists a polynomial time algorithm for closed tree queries. The problem remains open and deserves further investigation

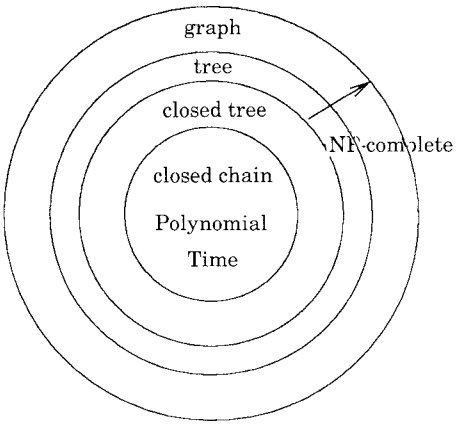
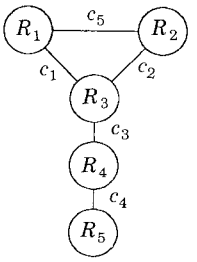


Fig. 8. Complexity of problem MRP for various queries. Note that the problem of finding the best join strategy for queries whose closure is a tree (closed trees) remains open



$c_1 = \{R_1.A, R_3\ C\}$
$c_2 = \{R_2.B, R_3\ C\}$
$c_3 = \{R_3\ D, R_4.E\}$
$c_4 = \{R_4\ F, R_5.G\}$
$c_5 = \{R_1\ A, R_2\ B\}$

Fig. 9. Initial query graph for heuristics examples

Table III Data for the Heuristic Examples

i	1	2	3	4	5
			Cardinality of Relation		
$\lvert R_i \rvert$	10	60	95	30	50
			Tuple Size		
$\lvert t_{R_i} \rvert$	4	4	3	1	5

Note. Initially, part(R_i) = X, $1 \leq i \leq 5$ Selectivity factor between any two relations is 0 02.

apply the algorithms developed in this section to construct a clause string from the graph. The behavior of the heuristics is analyzed in Section 7. One of our heuristics is a consistent generalization of algorithm CHAIN in the sense that an optimal solution will be found if q is a closed chain.

For each algorithm, a numerical example and graphical description are given based on the query graph in Figure 9 and data presented in Table III.

6.1 Kruskal Heuristic

The Kruskal heuristic (KH) is a greedy strategy similar to Kruskal's algorithm [20] for finding a minimum cost spanning tree from a weighted graph. Each time a lowest cost clause, say $\{R.A, S.B\}$, is selected; the two nodes R

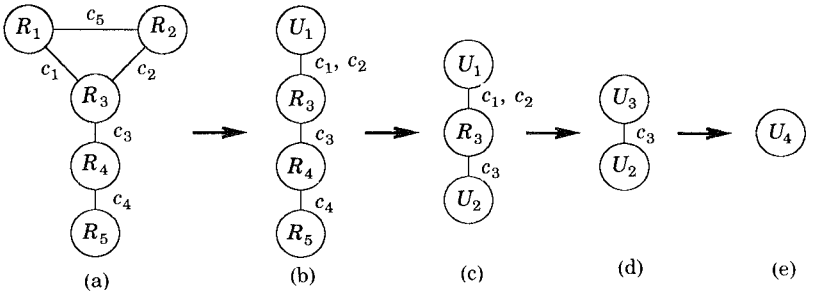


Fig. 10. Query graph reduction by algorithm KH.

and S are then collapsed to a single node U. Since U represents the join result of R and S, its cardinality with be $|R| \times |S| \times \alpha$, where α is the selectivity factor between R and S, and the width of its tuples is $|t_R| + |t_S|$. The partition attributes of U are calculated as follows:

(1) If R needs to be repartitioned (such a repartitioning destroys the original partition attributes of R), set newpart(R) to $\{A\}$; otherwise newpart(R) = part(R).
(2) Similarly, if S needs to be repartitioned, set newpart(S) to $\{B\}$; otherwise newpart(S) = part(S).
(3) part(U) = newpart(R) $\cup$ newpart(S).

The algorithm repeats the above steps until the query graph becomes a single node.

Since clauses on an edge share common relations, it is likely that many clauses have the same minimum cost. The essence of the algorithm is to assign a *number value* to each clause $\{R.A, S.B\}$, where number($\{R.A, S.B\}$) is defined as the total number of clauses on distinct edges (after collapsing R and S) that have $R.A$ or $S.B$ as a component attribute. Among the clauses with the same minimum cost, the algorithm selects among those with the largest number value. Performing a join on such a clause is good, because its two component attributes become the partition attributes of the new node and other clauses may be able to exploit them.

Example 4. Figure 10 shows the application of algorithm KH to the query graph in Figure 9. First, since both $\{R_1.A, R_2.B\}$ and $\{R_4.F, R_5.G\}$ have the lowest cost, 840, and number($\{R_1.A, R_2.B\}$) = 1 > number($\{R_4.F, R_5.G\}$) = 0, $\{R_1.A, R_2.B\}$ is selected. R_1 and R_2 are collapsed to U_1 (Figure 10b). We get $|U_1| = 12$; $|t_{U_1}| = 8$; part(U_1) = $\{A, B\}$. Next, $\{R_4.F, R_5.G\}$ is selected. R_4 and R_5 are collapsed to U_2 (Figure 10c). We get $|U_2| = 30$; $|t_{U_2}| = 6$; part(U_2) = $\{F, G\}$. Then, since both $\{R_1.A, R_3.C\}$ and $\{R_2.B, R_3.C\}$ have cost 951 and number value 0, we arbitrarily choose one,

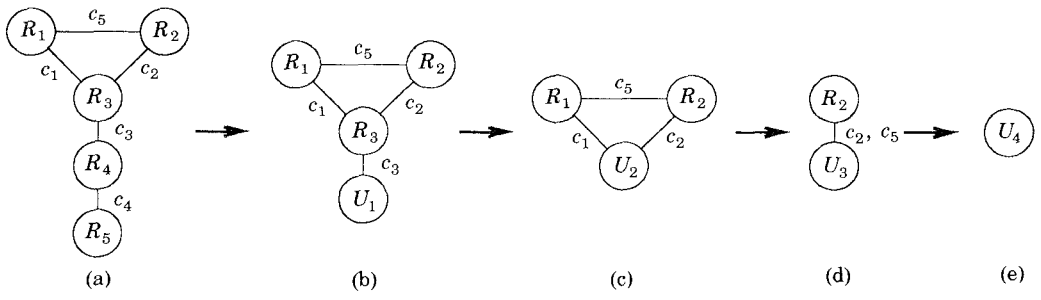


Fig. 11. Query graph reduction by algorithm PH.

say $\{R_1.A, R_3.C\}$. U_1 and R_3 are collapsed to U_3 (Figure 10d). We get $|U_3| \approx 22$; $|t_{U_3}| = 11$; part$(U_3) = \{A, B, C\}$. Finally, the clause $\{R_3.D, R_4.E\}$ with cost 1266 is selected. The string produced by the algorithm is $\{R_1.A, R_2.B\}\{R_4.F, R_5.G\}\{R_1.A, R_3.C\}\{R_3.D, R_4.E\}$ and the cost is 840 + 840 + 951 + 1266 = 3897.

6.2 Prim Heuristic

The Prim heuristic (PH) is similar to Prim's algorithm [23] for finding a minimum cost spanning tree from a weighted graph. At each step, a node in the graph is designated as a "pivot." (Heuristically, the initial pivot is a relation of smallest size.) The algorithm considers only the clauses that are on the edges connected to the pivot and selects a clause, say $\{R.A, S.B\}$, with the lowest cost and largest number value. R and S are then collapsed to a single node U. U becomes the new pivot, and its size and partition attributes are calculated as described previously. The algorithm repeats the above steps until the graph is reduced to a single node.

Example 5. Figure 11 shows the application of algorithm PH to the query graph in Figure 9. Initially, pivot = R_4. The clause $\{R_4.F, R_5.G\}$ with cost 840 is selected. R_4 and R_5 are collapsed to U_1 (Figure 11b). We get $|U_1| = 30$; $|t_{U_1}| = 6$; part$(U_1) = \{F, G\}$. The pivot becomes U_1. Next, $\{R_3.D, R_4.E\}$ with cost 1395 is selected. U_1 and R_3 are collapsed to U_2 (Figure 11c). We get $|U_2| = 57$; $|t_{U_2}| = 9$; part$(U_2) = \{D, E\}$; pivot = U_2. Then $\{R_1.A, R_3.C\}$ with cost 1659 is selected. U_2 and R_1 are collapsed to U_3 (Figure 11d). We get $|U_3| \approx 11$; $|t_{U_3}| = 13$; part$(U_3) = \{A, C\}$; pivot = U_3. Finally, since both $\{R_1.A, R_2.B\}$ and $\{R_2.B, R_3.C\}$ have cost 863 and number value 0, we arbitrarily choose one, say $\{R_1.A, R_2.B\}$. The string produced by the algorithm is $\{R_4.R, R_5.G\}\{R_3.D, R_4.E\}\{R_1.A, R_3.C\}\{R_1.A, R_2.B\}$ and the cost is 840 + 1395 + 1659 + 863 = 4757.

6.3 Hybrid Heuristics

A given graph may have several chains, that is, paths on which all nodes except the (distinct) endpoints (referred to as boundaries) are connected with

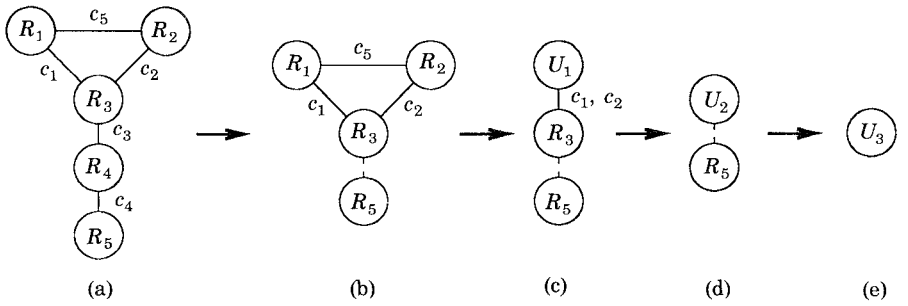


Fig. 12. Query graph reduction by algorithm HKH; dashed line represents generalized clause.

exactly two other nodes. It may be beneficial to process these chains based on algorithm CHAIN while applying the previous heuristics to the remaining part of the graph. This is also intuitively reasonable because by so doing, the optimal solution can be achieved when the given graph is a closed chain.

The algorithm first replaces each chain with a generalized clause. The component attributes of the generalized clause are the partition attributes of the boundaries after executing the chain's minimum cost string, and the cost of it is given by that of the string, which is calculated (by algorithm CHAIN) as a function of the sizes and partition attributes of the boundaries. Notice that, because it depends on the current sizes and partition attributes of a chain's boundaries, the generalized clause may vary during the execution of the algorithm (see the example below).

The algorithm then simulates algorithm KH or PH on the resulting graph. (The former is referred to as algorithm HKH, and the latter as HPH). During the course of execution, if a generalized clause is selected, its corresponding chain is collapsed to a single node. Since several chains may share the same two boundaries, collapsing one of them may cause the generalized clauses of the other chains to become self-loops in the query graph. When the algorithm detects this situation, it removes the loops, putting back nodes on their corresponding chains into the graph.

Example 6. Figure 12 shows the application of algorithm HKH to the query graph in Figure 9. First, the chain $\{\{R_3, R_4\}, \{R_4, R_5\}\}$ is replaced by the generalized clause $\{R_3.D, R_5.\perp\}$ with cost 2235, where $\perp$ indicates that after executing the minimum cost string $\{R_4.F, R_5.G\}\{R_4.E, R_3.D\}$, part($R_5$) is empty (Figure 12b). Next, the clause $\{R_1.A, R_2.B\}$ with cost 840 is selected. R_1 and R_2 are collapsed to U_1 (Figure 12c). We get $|U_1| = 12$; $|t_{U_1}| = 8$; part(U_1) = $\{A, B\}$. Then, since both $\{R_1.A, R_3.C\}$ and $\{R_2.B, R_3.C\}$ have cost 951 and number value 0, we arbitrarily pick one, say $\{R_2.B, R_3.C\}$. U_1 and R_3 are collapsed to U_2 (Figure 12d). We get $|U_2| \approx 22$; $|t_{U_2}| = 11$; part(U_2) = $\{A, B, C\}$. Since the size of one of the boundaries of the chain is changed, the generalized clause now becomes $\{R_3.\perp, R_5.G\}$ with cost 2034. Finally, U_2 and R_5 are collapsed to U_3 (Figure 12e). The

string produced by the algorithm is $\{R_1.A, R_2.B\}\{R_2.B, R_3.C\}\{R_3.D, R_4.E\}\{R_4.F, R_5.G\}$ and the cost is 840 + 951 + 2034 = 3825.

7. PERFORMANCE ANALYSIS

In this section we discuss experiments to compare the results produced by each heuristic with the optimums obtained from exhaustive search and evaluate the relative performance of these heuristics.

7.1 Experimental Parameters

Parameters used in the experiments can be classified into two categories: those related to databases and those specific to a query graph.

(1) *Database-Dependent Parameters*

$\|R\|$	Number of tuples in relation R
$\|t_R\|$	Width (in bytes) of a tuple in R
part(R)	Initial partition attribute of R
$\alpha_{R.S}$	Join selectivity factor between relations R and S.

(2) *Query-Dependent Parameters*

$\|QG\|$	Size (in number of nodes) of a query graph QG
$\|cn\|$	Size (in number of nodes) of a chain in the graph (including boundaries)
Num	Number of chains in the graph
$\|$clauses(e, QG)$\|$	Number of clauses on edge e in the graph.

To construct a query graph, we used a random-number generator to produce edges between nodes and to choose nodes in chains. Clauses on edges were also generated randomly and join attributes were drawn randomly from the range A to D. Such a range was chosen because in actual applications, relations generally contain no more than this number of join attributes, even in a fairly complex query.[11] After generating a query graph, we used its closure as a test graph for the algorithms.

In all the experiments presented here, the constants α and β for processing cost and repartitioning cost (cf. Definition 2) were fixed at 1 and 2, respectively. In fact, the constants have little effect on the relative behavior of our algorithms. In further experiments, we ran the algorithms with various constants (e.g., with $\alpha = 1$, $\beta = 10$; $\alpha = 10$, $\beta = 1$, etc.). It was found that both the relative performance of the heuristics and their performance relative to optimums are insensitive to these parameters.

7.2 Heuristic Performance Relative to Optimal Value

Suppose there are m spanning trees for a given graph, named $T_1, T_2, \ldots, T_m$, and edges in T_j, $1 \le j \le m$, are denoted by e_i^j, $1 \le i \le |QG| - 1$. The

[11] Part(R) used in the experiments was drawn randomly from the range A to G, as relations may not be partitioned on any of their join attributes initially.

Table IV. Heuristic Performance/Optimal Performance

	KH		PH		HKH		HPH	
Size of graph	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
6	1.04	0.00628	1.09	0.01292	1.02	0.01314	1.06	0.00823
7	1.05	0.01034	1.10	0.01125	1.04	0.00909	1.07	0.00931
8	1.07	0.01045	1.09	0.01296	1.05	0.00825	1.08	0.00911
9	1.06	0.01172	1.10	0.01325	1.04	0.00877	1.08	0.01175
10	1.09	0.01188	1.12	0.01475	1 08	0.00984	1.11	0.00928
11	1.08	0.01081	1.11	0.01407	1.07	0.00866	1.09	0.01318
12	1.10	0.01562	1.13	0.01368	1.08	0.02182	1.11	0.01236

Notes. (1) Query parameters: Number of chains = 1; chain size = $k \times$ graph size, where k was drawn from [1/2, 2/3]; number of clauses on an edge was drawn from [1, 3]

(2) Database parameters: Relation's cardinality was drawn from $[1 \times 10^a, 2 \times 10^a]$, where a can vary; tuple's width was drawn from [1, 10]; selectivity factor from $[0.1 \times 10^{-a}, 1 \times 10^{-a}]$; part($R$) from $[A, G]$.

number of clause strings that need to be examined by exhaustive search is

$$\prod_{j=1}^{m} \left\{ \left(\prod_{i=1}^{|QG|-1} |\text{clauses}(e_i^j, QG)| \right) \times (|QG| - 1)! \right\}.$$

As can be seen from the formula, the price paid for examining all possible strings becomes prohibitively high when the size of a graph is large. To keep the experiments manageable, the data were drawn from the following moderate ranges.[12] $|QG|$ was ranged from 6 to 12, *Num* was fixed at 1, $|cn|$ was drawn from the range [1/2, 2/3] of $|QG|$, $|R|$ was drawn from the range $[1 \times 10^a, 2 \times 10^a]$, $|t_R|$ from the range [1, 10], and $|\text{clauses}(e, QG)|$ from the range [1, 3] for any edge e in the graph. The selectivity factor between each two relations was drawn from the range $[0.1 \times 10^{-a}, 1 \times 10^{-a}]$. Special care was taken to generate the selectivity factors to ensure that joining nodes in different order yields the same size result.

The experimental results are summarized in Table IV. Each mean in the table represents the average value of the relative costs obtained by applying an algorithm to ten test graphs. A relative cost is the quotient of the solution's cost of the algorithm divided by the optimum. The means show that all the algorithms have satisfactory performance for random spanning trees and low selectivity factors, and in many situations, algorithms KH and HKH seem a bit superior to the other two algorithms. In addition, the small variances for each algorithm indicate that the algorithms exhibit consistent behavior.

7.3 Effect of Database Parameters

The purpose of this subsection is to analyze the effect of varying the database-dependent parameters on the relative performance of the heuristic

[12] In the sequel, "a parameter was drawn from the range $[a, b]$" actually means that the parameter was drawn randomly from a uniform distribution with the range $[a, b]$.

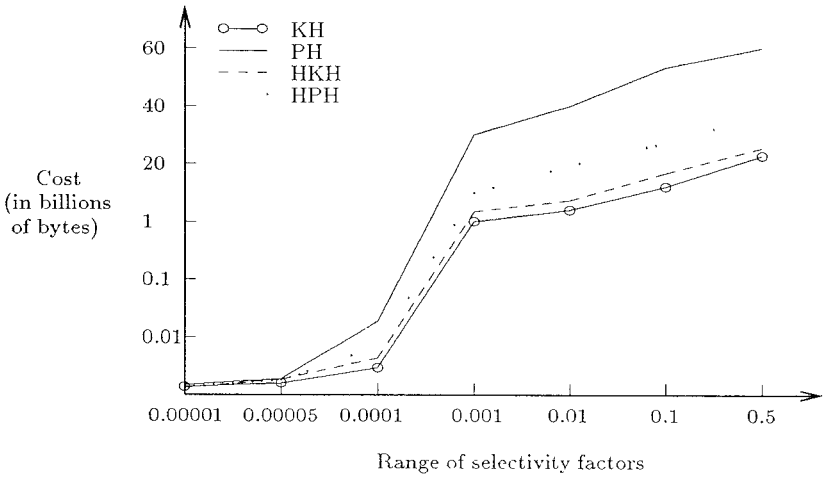


Fig. 13. Effect of selectivity factors (relation's cardinality was drawn from [10,000, 20,000], tuple's width from [1, 10], graph size = 12, chain numbers = 2, chain size = 4, number of clauses on an edge = 2)

procedures. To avoid the mutual influence of parameters, the analysis was carried out by fixing the parameters related to query graphs, which were set as follows: $|QG| = 12$, $Num = 2$, $|cn| = 4$, $|\text{clauses}(e, QG)| = 2$ for any edge e in QG. The chains have been included in order to distinguish the simple heuristics from their hybrid counterparts.

Figure 13 illustrates the behavior of the algorithms for varying join selectivity factors.[13] Examining the figure, we see that all the heuristics have close behavior for low selectivity factors ($\alpha_{R,S} \leq 0.00005$).

Notice also that algorithm PH deteriorates significantly as the selectivity factors increase. When selectivity factors are high, the more nodes are joined, the larger the pivot becomes. Consequently, joining the pivot with even more nodes will incur an intolerable cost.

Figure 14 shows the effect of varying the relation sizes on the relative performance of the algorithms. The figure shows clearly that algorithm PH becomes unattractive when the sizes of relations increase.

7.4 Effect of Query Parameters

The main distinction between the hybrid heuristics and simple ones lies in the way they handle chains. The objective of this subsection is to explore the effect of varying chains in a query graph on the behavior of the heuristics. The following parameters' values were assumed throughout the experiments: $|R|$ was drawn from the range [100, 1,000,000], $|t_R|$ was drawn from [1, 10], $\alpha_{R,S}$ was fixed at 0.1, and $|\text{clauses}(e, QG)|$ was fixed at 2.

[13] In subsequent experiments, ten query graphs were tested for each algorithm and the average value of the solutions' costs produced by each algorithm was plotted.

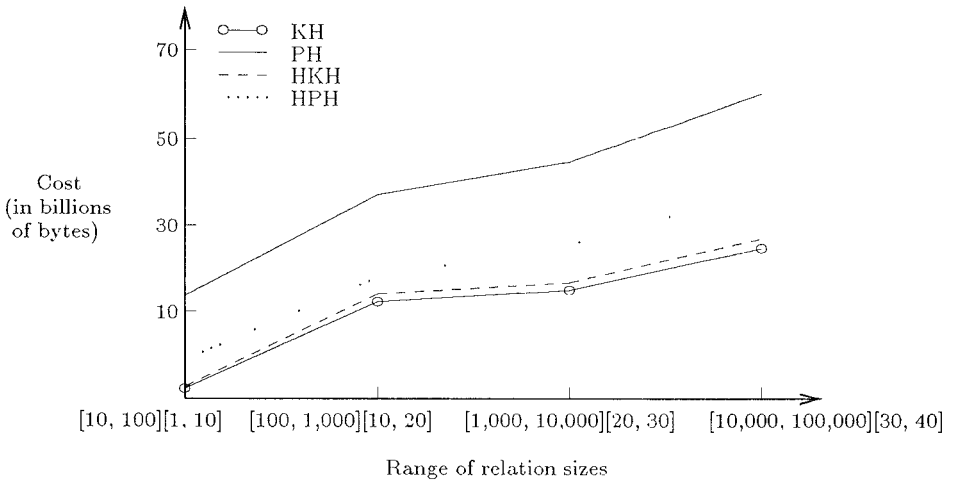


Fig. 14. Impact of relation size, where relation size = relation's cardinality × tuple's width (the first item of each label on the x axis gives the range of relation's cardinality and the second item gives the range of tuple's width; selectivity factor = 0.1, graph size = 12, chain numbers = 2, chain size = 4, number of clauses on an edge = 2).

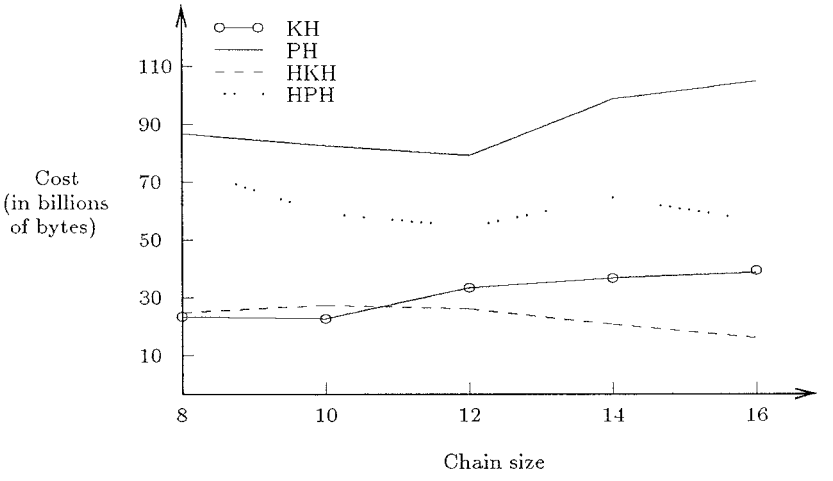


Fig. 15. Cost of heuristics as a function of chain size (graph size = 20, number of chains = 1, selectivity factor = 0.1, number of clauses on an edge = 2, relation's cardinality was drawn from [100, 1,000,000], tuple's width from [1, 10]).

Figure 15 shows the effect of changing the chain size in a graph, where the size of the graph was fixed at 20 and *Num* at 1. It is apparent that as the chain becomes a major portion of a graph, the hybrid heuristics become better than the simple ones. One expects this because the hybrid heuristics guaran-

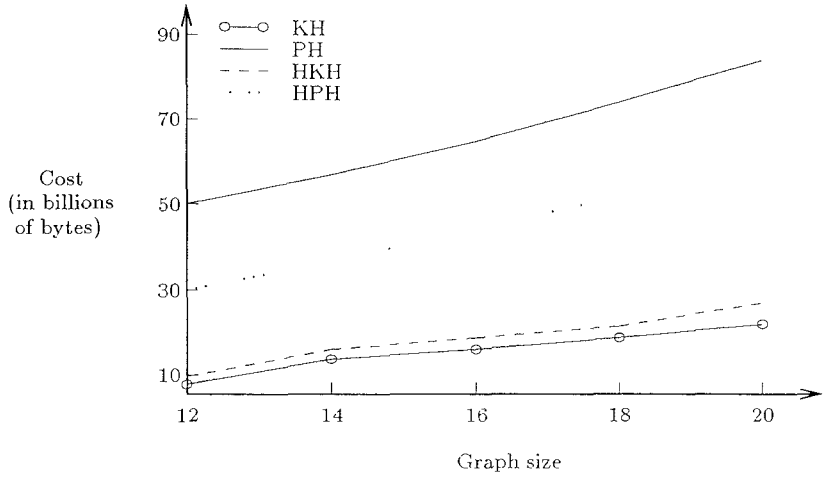


Fig. 16 Cost of heuristics as a function of graph size (chain size = 1/2 × graph size, number of chains = 1, selectivity factor = 0.1, number of clauses on an edge = 2, relation's cardinality was drawn from [100, 1,000,000], tuple's width from [1, 10])

tee that an optimal solution can be achieved on a chain. The poor performance of algorithm PH when the chain size becomes large is due to the way it picks clauses, which is limited by its inherently local view. This situation tends to be worse if it starts searching inside a chain—very few choices can be made in determining which of the clauses should be picked at each step.

On the other hand, the size of a graph has very little impact on the performance of the heuristics. Figure 16 illustrates the effect of varying the size of a graph while fixing the portion of the chain in it (this portion was fixed at 1/2). It is clear that increasing only the graph size does not affect the relative performance of the heuristics—the gaps between these algorithms remain and gradually become larger.

Figure 17 shows the effect of varying the number of chains in a graph (where $|QG|$ was fixed at 20 and $|cn|$ at 4). Although the number of chains also influences the performance of the hybrid heuristics, it has a less significant effect than the size of a chain. The big gap between algorithms HPH and KH (HKH) may be due to the fact that the chosen size of chains is too small, as compared with that of the whole graph. Notice that there is a slight increase of the gap between algorithms HKH and KH when the number of chains is 4. This indicates that the existence of many short chains in a graph can have a negative effect on the performance of algorithm HKH. A possible explanation for this is that these chains decrease the number of nonchain edges in the graph and consequently restrict the total number of choices that can be made when globally picking a clause.

In conclusion, when selectivity factors are low, the proposed heuristics have close performance. In the presence of high selectivity factors, algorithm PH becomes less competitive while algorithm KH (and HKH) remain good. If

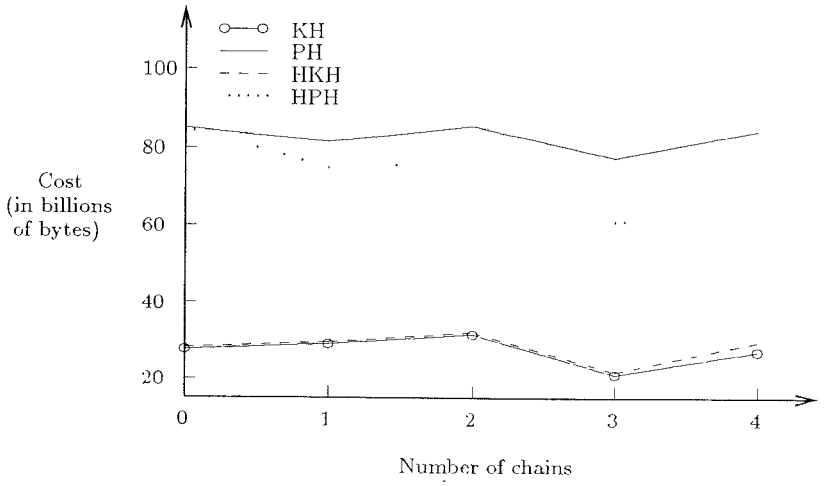


Fig. 17. Cost of heuristics as a function of number of chains (graph size = 20, chain size = 4, selectivity factor = 0.1, number of clauses on an edge = 2, relation's cardinality was drawn from [100, 1,000,000], tuple's width from [1, 10]).

a graph contains long chains (with size being over half of the number of nodes in the graph), we suggest to use the hybrid Kruskal heuristic (HKH).

8. SUMMARY

In this paper, we investigated ways of minimizing response time for various important queries in loosely coupled multiprocessor systems using a hash-partitioned data distribution scheme. First, we developed a dynamic programming algorithm for closed chain queries. Next, we proved the NP-completeness for more general queries and proposed four heuristics for them. Our simulation results show that an algorithm similar to Kruskal's spanning tree algorithm performs well when chains in a query graph are small. When chains are long, then a hybrid algorithm using the chain algorithm with Kruskal's is best.

Like other relevant work [15, 25], the heuristics presented in the paper relied on the knowledge of selectivity factors, which were used to find out the sizes of intermediate results. This information, however, is not known a priori in actual environments.[14] Fortunately, our qualitative conclusions were independent of the selectivity factors.

Our results have all assumed that relations were partitioned on a single attribute. Generalizing beyond this is not difficult. 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

[14]A practical approach would be to estimate the selectivity factors like those used in most real systems (see, e.g., [26]). Readers may also refer to Gardy and Puech [13] and Lipton and Naughton [21] for theoretical analyses of the sizes of intermediate relations.

$\{R.A, S.C\}$, treating $\{R.B, S.D\}$ as a selection condition. On the other hand, if the join were based on $\{R.A, S.C\}$ and R were partitioned on AB and S on CD, both relations would have to be repartitioned to A and C, respectively. Therefore, the set of repartitionings required when R is partitioned on A alone is always a subset of those required when R is partitioned on AB. To handle multiattribute partitions, we can treat the partition attributes (e.g., AB) as if they were a single attribute. Thus, to perform the join having clauses $\{R.A, S.C\}$, $\{R.B, S.D\}$, and $\{R.E, S.F\}$, given AB and CD as partition attributes, we process $\{\{R.A, S.C\}, \{R.B, S.D\}\}$ and use $\{R.E, S.F\}$ as a selection condition.

We have not discussed the possibility of using multiple copies of relations, perhaps partitioned on different attributes. Often, keeping multiple copies at each node reduces the cost of repartitionings considerably. As an example, consider again the suppliers-and-parts database and SQL query presented in Section 1. Suppose one copy of SUP is partitioned on s#, one copy of PART on p#, one copy of SUP-PART on s#, and another copy of SUP-PART on p#. When repartitioning T_2 to SUP-PART.p#, instead of sending its complete tuples, the second strategy could send the *surrogates* [8] of tuples in SUP-PART, along with the complete tuples in SUP. These surrogates would then be materialized at each node by consulting the local fragment of SUP-PART that is partitioned on p#. Suppose sending surrogates takes 10 seconds. Then the cost of this scheme would be 100 + 10 + 100 = 210 seconds, which achieves a significant improvement over the original scheme. Investigating how queries can be optimized in such a highly parallel environment with replicated data seems to be a very interesting problem for future research.

ACKNOWLEDGMENTS

The authors are deeply indebted to the anonymous referees and the editor, Won Kim, for their encouragement and thoughtful recommendations. They also wish to thank Richard Cole, Zvi Kedem, Bud Mishra, and Paul Spirakis for helpful discussions in the formative stages of this work.

REFERENCES

1. Apers, P. M. G., Hevner, A. R., and Yao, S. B Optimization algorithms for distributed queries. *IEEE Trans. Softw. Eng. SE-9*, 1 (Jan. 1983), 57-68.
2. Babb, E. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst. 4*, 1 (Mar. 1979), 1-29.
3. Bernstein, P. A , Goodman, N., Wong, E , Christopher, L. R. and Rothnie, J. B., Jr. Query processing is a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 602-625.
4. Bitton, D., Boral, H , DeWitt, D. J., and Wilkinson, W K. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst. 8*, 3 (Sept 1983) 324-353.
5. Chen, A. L. P , and Li, V. O. K. An optimal algorithm for processing distributed star queries. *IEEE Trans. Softw. Eng. SE-11*, 10 (Oct. 1985), 1097-1107.
6. Chiu, D. M., Bernstein, P. A., and Ho, Y. C. Optimizing chain queries in a distributed database system. *SIAM J. Comput. 13*, 1 (Feb. 1984), 116-134
7. Chiu, D. M., and Ho, Y. C. A methodology for interpreting tree queries into optimal

semi-join expressions. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (May 1980). ACM, New York, 1980, pp. 169-178.

8. CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec. 1979), 397-434.
9. COPELAND, G., ALEXANDER, W., BOUGHTER, E., AND KELLER, T. Data placement in Bubba. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (1988). ACM, New York, 1988, pp. 99-108.
10. DEWITT, D. J., AND GERBER, R. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases* (Stockholm, Aug. 1985), pp. 151-164.
11. DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. GAMMA—A high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Aug. 1986), pp. 228-237.
12. DEWITT, D. J., SMITH, M., AND BORAL, H. A single user performance evaluation of the Teradata Database Machine. MCC Tech. Rep. DB-081-87, 1987.
13. GARDY, D., AND PUECH, C. On the effect of join operations on relation sizes. *ACM Trans. Database Syst. 14*, 4 (Dec. 1989), 574-603.
14. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, Calif., 1979.
15. GAVISH, B., AND SEGEV, A. Set query optimization in distributed database systems. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 265-293.
16. GOODMAN, N , AND SHMUELI, O. Tree queries: A simple class of relational queries *ACM Trans. Database Syst. 7*, 4 (Dec. 1982), 653-677.
17. KIM, W. A new way to compute the product and join of relations. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (Santa Monica, Calif., May 1980). ACM, New York, 1980, pp. 179-187.
18. KITSUREGAWA, M., TANAKA, H., AND MOTO-OKA, T. Relational algebra machine GRACE. In *RIMS Symposium on Software Science and Engineering* (1982). *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1983, pp. 191-212.
19. KITSUREGAWA, M., ET AL. Application of hash to data base machine and its architecture. *New Generation Comput. 1* (1983), 62-74.
20. KRUSKAL, J. B., JR. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc. 7*, 1 (1956), 48-50.
21. LIPTON, R. J., AND NAUGHTON, J. F. Query size estimation by adaptive sampling. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Apr. 1990). ACM, New York, 1990, pp. 40-46.
22. PRAMANIK, S., AND VINEYARD, D. Optimizing join queries in distributed databases. *IEEE Trans. Softw. Eng. 14*, 9 (Sept. 1988), 1319-1326.
23. PRIM, R. C. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.* (1957), 1389-1401.
24. SACCO, G. M. Fragmentation: A technique for efficient query processing. *ACM Trans. Database Syst. 11*, 2 (Jun. 1986), 113-133.
25. SEGEV, A. Optimization of join operations in horizontally partitioned database systems. *ACM Trans. Database Syst. 11*, 1 (Mar. 1986), 48-80.
26. SELINGER, P. G., ASTRAHAN, M M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database system. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (Boston, Mass., May 1979). ACM, New York, 1979, pp 23-34.
27. SHAPIRO, L. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 239-264.
28. SHASHA, D. Query processing in a symmetric parallel environment. In *Proceedings of the Advanced Database Symposium* (Tokyo, Japan, Aug. 1986), pp. 183-192.
29. SHASHA, D., AND SPIRAKIS, P. Fast parallel algorithms for processing of joins. In *Proceedings of the International Conference on Supercomputing* (Athens, Greece, June 1987).

30 STAMOS, J. W., AND YOUNG, H. C A symmetric fragment and replicate algorithm for distributed joins. Res. Rep RJ7188, IBM Corp., San Jose, Dec 1989.
31. STONEBRAKER, M., AND NEUHOLD, E. A distributed database version of INGRES In *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks* (May 1977).
32. SUN, W., MENG, W , AND YU, C. Query optimization in object-oriented database systems Unpublished manuscript, Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, 1990.
33 TERADATA CORPORATION. Database computer system concepts and facilities. Document C02-0001-01, Teradata Corporation, Los Angeles, Oct. 1984
34. VALDURIEZ, P., AND GARDARIN, G. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 133–161.
35 WILLIAM, R., ET AL. R*: An overview of the architecture Res. Rep. RJ3325, IBM Corp., San Jose, Dec. 1981.
36. WONG, E. Dynamic rematerialization: Processing distributed queries using redundant data *IEEE Trans Softw. Eng. SE-9*, 3 (May 1983), 228–232.
37 YU, C. T., OZSOYOGLU, M Z., AND LAM, K. Distributed query optimization for tree queries *J. Comput. Syst. Sci. 29* (1984), 409–445.
38. YU, C. T., GUH, K. C., ZHANG, W., TEMPLETON, M., BRILL, D., AND CHEN, A. L. P Algorithms to process distributed queries in fast local networks *IEEE Trans. Comput. C-36*, 10 (Oct 1987), 1153–1164.

Received January 1988; revised May 1989 and April 1990; accepted April 1990