Query Processing in a System for Distributed Databases (SDD-1)

PHILIP A. BERNSTEIN and NATHAN GOODMAN
Harvard University
EUGENE WONG
University of California at Berkeley
CHRISTOPHER L. REEVE
Massachusetts Institute of Technology
and
JAMES B. ROTHNIE, Jr.
Computer Corporation of America

This paper describes the techniques used to optimize relational queries in the SDD-1 distributed database system. Queries are submitted to SDD-1 in a high-level procedural language called Datalanguage. Optimization begins by translating each Datalanguage query into a relational calculus form called an *envelope*, which is essentially an aggregate-free QUEL query. This paper is primarily concerned with the optimization of envelopes.

Envelopes are processed in two phases. The first phase executes relational operations at various sites of the distributed database in order to delimit a subset of the database that contains all data relevant to the envelope. This subset is called a *reduction* of the database. The second phase transmits the reduction to one designated site, and the query is executed locally at that site.

The critical optimization problem is to perform the reduction phase efficiently. Success depends on designing a good repertoire of operators to use during this phase, and an effective algorithm for deciding which of these operators to use in processing a given envelope against a given database. The principal reduction operator that we employ is called a *semijoin*. In this paper we define the semijoin operator, explain why semijoin is an effective reduction operator, and present an algorithm that constructs a cost-effective program of semijoins, given an envelope and a database.

Key Words and Phrases: distributed databases, relational databases, query processing, query optimization, semijoins
CR Categories: 3.70, 4.33

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract N00039-77-C-0074, ARPA Order 3175-6.

Authors' addresses: P.A. Bernstein and N. Goodman, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138; E. Wong, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720; C.L. Reeve, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; J.B. Rothnie, Jr., Computer Corporation of America, 675 Massachusetts Avenue, Cambridge, MA 02139.

1. INTRODUCTION

SDD-1 is a distributed database system developed by the Computer Corporation of America [23]. SDD-1 permits a relational database to be distributed among the sites of a computer network, yet accessed as if it were stored at a single site. Users interact with SDD-1 by submitting queries coded in a high-level procedural language called Datalanguage [20]. Figures 1 and 2 illustrate an SDD-1 database and a Datalanguage query. This paper is concerned with efficient execution of such queries. Other aspects of SDD-1 are discussed in [5, 6, 14, 23].

Our objective is to process queries with a minimum quantity of intersite data transfer. That is, we assume network bandwidth to be the system bottleneck and seek to minimize use of this resource; all other resources are assumed to be free.[1] This assumption is appropriate in SDD-1 because the network is the slowest system component by two orders of magnitude.[2] This assumption has been

Database D

S(s#,	name,	location)	Y(s#,	p#,	qty)	P(p#,	name,	type)
1,	Acme,	MA	1,	1,	20	1,	LSI,	micro
2,	Best,	MA	1,	2,	50	2,	P11,	mini
3,	Mid,	NY	3,	3,	50	3,	360,	main
4,	Nadir,	CA	4,	1,	10	4,	CRI,	huge
			4,	5,	75	5,	8080,	micro

S describes suppliers.
P describes parts.
Y tells which suppliers supply which parts and in what quantity.

Assume that S, Y, and P are sorted at sites 1, 2, and 3, respectively.

Fig. 1. Example database.

Description of query
List the supplier name, part name, and quantity supplied for all parts supplied by a Massachusetts supplier. Also, print how many of these are minis.

Query Q

```
Begin
  Count := 0;
  For S
    If S.location = "MA" then for Y
      If S.s# = Y.s# then for P
        If Y.p# = P.p# then begin
          Print S.name, P.name, Y.qty;
          If P.type = "mini" then Count := Count + 1; end;;;
  Print "Number of minis is", Count ;
End.
```

Fig. 2. Example datalanguage query. Datalanguage used in this example is defined in the appendix.

[1] In practice, database processing within sites is considered as a secondary objective. For expository clarity, we shall not treat this issue.

[2] Sites in SDD-1 are mainframe computers (PDP-10s), while the network is a packet-switched long-distance network (Arpanet). Sustainable bandwidth on the network is at most 10 kbits per second (see [24–26]).

Envelope E

Retrieve (S.s#, S.name, S.location)	where *qualification*
Retrieve (Y.s#, Y.p#, Y.qty)	where *qualification*
Retrieve (P.p#, P.name, P.type)	where *qualification*

qualification: S.location = "MA" ∧ S.s# = Y.s# ∧ Y.p# = P.p#

Fig. 3. Envelope for query of Figure 2. Envelopes are defined in Section 2. Intuitively, an envelope specifies a subset of each relation in the database. We express envelopes in a relational calculus similar to QUEL [15].

The result of envelope E is to retrieve *any superset* of the data specified by E. For example,

S(s#,	name,	location)	Y(s#,	p#,	qty)	P(p#,	name,	type)
1,	Acme,	MA	1,	1,	20	1,	LSI,	micro
2,	Best,	MA	1,	2,	50	2,	P11,	mini
						3,	360,	main
						4,	CRI,	huge
						5,	8080,	micro

The specific superset retrieved is determined by efficiency considerations.

The retrieved relations are also transmitted to a single site, for example, site 3.

Fig. 4. Processing envelope of Figure 3.

adopted by other researchers [7, 8, 13, 16, 17, 33, 34], although naturally it is not appropriate in every system [10, 19, 27]. Section 5 discusses the impact of this assumption on our approach.

Our algorithm has three main steps. Step 1 maps a Datalanguage query Q into a relational calculus form (an *envelope*) that specifies a *superset* of the database needed to answer Q (see Figure 3). Step 1 depends on details of Datalanguage and is of general interest only insofar as Datalanguage resembles other procedural query languages. This step is described in [11].

Step 2 *evaluates the envelope.* This step retrieves a *superset* of the database specified by the envelope, assembling the result at a single site S_a (see Figure 4). (The specific superset retrieved and the "assembly site" S_a are determined by efficiency considerations.) Step 2 is accomplished by translating the envelope into a program P containing relational operations (a *reducer*), followed by commands to move the results of P to S_a (see Figure 5). The goal is to construct a reducer P and select a site S_a such that the cost of computing P and moving the results to S_a is minimum over all reducers and sites. This optimization problem constitutes the core of the SDD-1 query processing algorithm and is the focus of this paper.

Step 3 executes Q at S_a using the data assembled by Step 2. Since Step 3 only involves local query processing, it will not be discussed further. Steps 1–3 are outlined in Figure 6.

The paper has five sections. Section 2 defines envelopes and the operations used to process envelopes. Section 3 presents techniques for estimating the cost and effect of a reducer composed of these operations. Section 4 presents a heuristic algorithm that compiles envelopes into efficient (though not necessarily optimal) reducers. Section 5 discusses related work and suggests directions for

```
Program P
1. S := S[location = "MA"]     ; restrict S to MA suppliers
2. Y := Y⟨s# = s#]S            ; this operation is semijoin
                               —it computes the set of Y
                               tuples that corresponds to
                               MA suppliers.
end
```

Figure 4 shows the result of applying P to the database of Figure 1.

Fig. 5. Program for envelope of Figure 3.

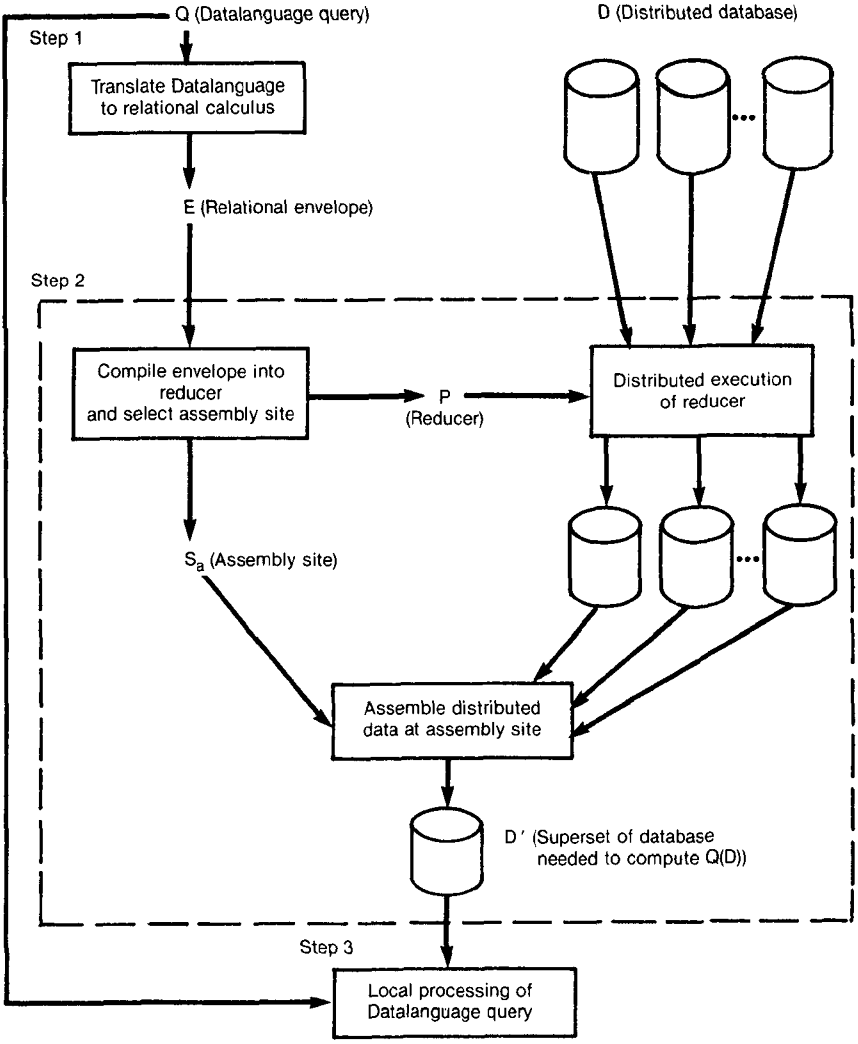

Q (Datalanguage query)
D (Distributed database)
Step 1
Translate Datalanguage to relational calculus
E (Relational envelope)
Step 2
Compile envelope into reducer and select assembly site
P (Reducer)
Distributed execution of reducer
S_a (Assembly site)
Assemble distributed data at assembly site
D′ (Superset of database needed to compute Q(D))
Step 3
Local processing of Datalanguage query

Fig. 6. Main steps of query processing algorithm.

(a) Relational Data Objects

Term	Definition
Domain	A set of values
Attribute	An alternate name for a domain
Relation schema	A description of a relation, consisting of a relation name and list of attributes
Relation	A subset of the Cartesian product of the domains of the attributes of the corresponding relation schema
Tuple	An element (or row) of a relation
Database	A set of relations
attr(R)	Attributes of relation R

(b) Relational Algebraic Operations

Restriction:	$R[A = k] = \{r \in R \mid r.A = k\}$ where r.A is the value of the A-domain in tuple r
Also	$R[A = B] = \{r \in R \mid r.A = r.B\}$
Projection:	$R[A_1, A_2, \ldots, A_n]$ $= \{\langle r.A_1, r.A_2, \ldots, r.A_n\rangle \mid r \in R\}$
Join:	$R[A = B]S = \{\langle r, s\rangle \mid r \in R, s \in S, \text{and } r.A = s.B\}$
Semijoin:	$R\langle A = B]S = (R[A = B]S)\,[\mathrm{attr}(R)]$ $= \{r \mid r \in R \wedge r.A \in s[B]\}$

Fig. 7. Relational terminology.

future research. We assume reader familiarity with relational databases at the level of [9]. A review of relational terminology appears in Figure 7.

2. QUERY PROCESSING STRATEGY

2.1 Envelopes

The attributes of relation R are denoted attr(R). Relation R_i' is a *subrelation* of relation R_i, if $\mathrm{attr}(R_i') \subseteq \mathrm{attr}(R_i)$ and $R_i' \subseteq R_i[\mathrm{attr}(R_i')]$. Let $D = \{R_1, \ldots, R_n\}$ and $D' = \{R_1', \ldots, R_n'\}$ be databases. D′ is a *subdatabase* of D, denoted $D' \leq D$, if R_i' is a subrelation of R_i for $i = 1, \ldots, n$. An *envelope* is a relational calculus expression that maps a database into a subdatabase. We express envelopes in a language similar to QUEL [15].

An *envelope* E consists of a *qualification* q and *target lists* $t_1, \ldots, t_n$. The term q is a Boolean formula with clauses of the form $R_i.A = R_j.B$ or $R_i.A = k$.[3] The terms $R_i.A$ and $R_j.B$ are called *indexed variables.* Each t_i is a set of variables indexed by R_i: that is, t_i is of the form $\{R_i.A_{i1}, \ldots, R_i.A_{il}\}$. Envelope E maps database D into subdatabase D′ defined by the following collection of QUEL queries.

Retrieve into $R_1'(t_1)$ where q.

•
•
•

Retrieve into $R_n'(t_n)$ where q.

We limit the form of envelopes in two additional ways. One, qualifications are

[3] Note that we avoid tuple variables. These can be accommodated by (conceptually) duplicating a relation, thereby having two relation names ranging over the same relation.

Join graph for envelope of figure 1.

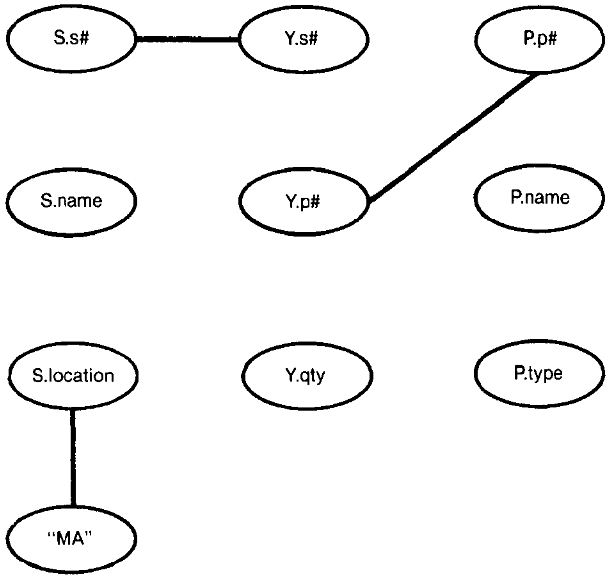


Fig. 8. Join graphs.

assumed to be pure conjunctions; disjunction is handled by placing the qualification in disjunctive normal form and treating each conjunct separately. Two, if $R_i.A$ is a term of q, then t_i must contain $R_i.A$.

E is *an envelope for* Datalanguage query Q if for all databases D, Q(E(D)) = Q(D). Intuitively, an envelope for Q "envelopes" or delimits the portions of the database needed to answer Q. In general, there are many envelopes for a given Q; a good envelope is one that tightly delimits the data needed by Q. Finding good envelopes is an optimization problem that depends on details of Datalanguage, and our approach to this problem is described in [11].

A graph representation of qualifications (a *join graph*) is useful. The nodes of a join graph represent indexed variables and constants, and the edges represent clauses. A join graph contains the edge $\{R_i.A, R_j.B\}$ (respectively, $\{R_i.A, k\}$) iff the qualification contains $R_i.A = R_j.B$ (respectively, $R_i.A = k$) (see Figure 8). The connected components of a join graph characterize the clauses implied by the qualification: Let N and N′ be nodes of the join graph for q; q implies N = N′ iff N and N′ are in the same connected component. (Proofs appear in [2, 3]. Note that if N and N′ represent distinct constants, q is unsatisfiable.)

2.2 Reducers

A *reduction* of database D with respect to E is any D′ such that $E(D) \leq D' \leq D$. A *reducer* for E is a sequential program[4] P of relational operations such that for

[4] A reducer is executed as a parallel program, however (see [23]).

Given

S(s#,	name,	location)
1,	Acme,	MA
2,	Best,	MA

Y(s#,	p#,	qty)
1,	1,	20
1,	2,	50
3,	3,	50
4,	1,	10
4,	5,	75

P(p#,	name,	type)
1,	LSI,	micro
2,	P11,	mini
3,	360,	main
4,	CRI,	huge
5,	8080,	micro

•Y⟨s# = s#]S = Y(s#, p#, qty) = {Y tuples that correspond to a MA supplier}

s#,	p#,	qty
1,	1,	20
1,	2,	50

•P⟨p# = p#]Y = P(p#, name, type) = {parts that are supplied by some MA supplier}

p#,	name,	type
1,	LSI,	micro
2,	P11,	mini

•S⟨s# = s#]Y = S(s#, name, location) = {MA suppliers who supply anything}

s#,	name,	location
1,	Acme,	MA

•All of these semijoins are profitable. However, Y⟨p# = p#]P would not be profitable.

Fig. 9. Semijoin.

all databases D, P(D) is a reduction of D with respect to E. Given E and D, our optimization task is to construct a reducer P and select a site S_a such that the cost of computing P(D) and moving the results to S_a is minimum over all reducers and sites. A *reduction operation* for E is an operation that is permitted in a reducer for E. A reduction operation reduces the size of D by eliminating data not specified by E(D). The *benefit* of a reduction operation is the amount of data it eliminates; the *cost* is the amount of intersite data transfer required to compute the operation.

Restrictions and projections have zero cost and nonnegative benefit, and so every restriction and projection permitted by E should be included in every reducer for E. The projections permitted by E are $R_i[t_i]$, for i = 1, . . . , n. The restrictions permitted by E can be determined from its join graph: E permits $R_i[A = B]$ (respectively, $R_i[A = k]$) iff q implies $R_i.A = R_i.B$ (respectively, $R_i.A = k$) iff $R_i.A$ and $R_i.B$ (respectively, k) are connected in the join graph.

To reduce the database further, data from two or more relations must be combined. The obvious operation for this purpose is *join*. However, our algorithm uses an operation called *semijoin*, which we deem to be superior. A semijoin is "half of a join"; *the semijoin of relation R_i by relation R_j on clause $R_i.A = R_j.B$*, denoted $R_i\langle A = B]R_j$, equals the join of R_i and R_j on that clause projected back onto attr(R_i) (see Figure 9). (Notice that semijoin, unlike join, is asymmetric; that is, $R_i\langle A = B]R_j \neq R_j\langle B = A]R_i$. The former reduces R_i, while the latter reduces R_j.) As with restrictions, the semijoins permitted by E can be determined from its join graph: E permits $R_i\langle A = B]R_j$ and $R_j\langle B = A]R_i$ iff $R_i.A$ and $R_j.B$ are connected in the join graph.

We prefer semijoins to joins for three reasons. First, $R_i\langle A = B]R_j \subseteq R_i$, and so semijoins monotonically reduce the size of the database. By contrast, joins *can increase* the size of the database; in the worst case, $|R_i[A = B]R_j| = |R_i| * |R_j|$.

- Let D be the database

R_1(A,	B)	R_2(B,	C)	R_3(A,	C)	R_4(C,	D,	E,	F,	G,	H,	I)
0	1	1	1	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1	1
site 1		site 2		site 3					site 4			

- Let E be the envelope

  q: $R_1.A = R_3.A \wedge R_1.B = R_2.B \wedge R_2.C = R_3.C \wedge R_3.C = R_4.C$

  $t_i = \text{attr}(R_i)$ for $i = 1, \ldots, 4$.

- Using semijoins, the optimal evaluation of E(D) is to move R_1, R_2, and R_3 to site 4—that is, no semijoins should be used. This requires the transmission of 12 data items.
- Using joins, the optimal evaluation is

  $R_{12} := R_2[B = B]R_1$ at site 2—cost = 4
  $R_{123} := R_{12}[A = A \wedge C = C]$ at site 2—cost = 4
  Note that $R_{123} = \{\ \}$
  $R_4 := R_4[C = C]R_{123}$ at site 4—cost = 0.
  Total cost = 8.

Fig. 10. Bad case for semijoins. This example is adapted from [2].

Second, semijoins can be computed with less intersite data transfer than joins. To compute $R_i\langle A = B]R_j$, we need only transmit a projection of a relation (viz., $R_j[B]$), whereas to compute $R_i[A = B]R_j$ we must transmit an entire relation. Of course, the semijoin may also have less effect than the join, since $R_i\langle A = B]R_j$ only reduces R_i, whereas $R_i[A = B]R_j$ simultaneously reduces R_i and R_j. However, the third advantage of semijoins is that the "reductive effect" of any *single* join can be attained by *two* semijoins, usually at lower cost, as follows.

Let $R_{ij} = R_i[A = B]R_j$. The *reductive effect* of this join is its effect on R_i and R_j, namely, $R_{ij}[\text{attr}(R_i)]$ and $R_{ij}[\text{attr}(R_j)]$. By definition of projection,

$$\begin{aligned} R_{ij}[\text{attr}(R_i)] &= \{r_i \mid \exists \langle r_i, r_j\rangle \in R_{ij}\} \\ &= \{r_i \in R_i \mid (\exists\, r_j \in R_j)(r_i.A = r_j.B)\}, && \text{by definition of join} \\ &= R_i\langle A = B]R_j, && \text{by definition of semijoin.} \end{aligned}$$

Similarly, $R_{ij}[\text{attr}(R_j)] = R_j\langle B = A]R_i$. Thus the reductive effect of $R_i[A = B]R_j$ is attained by two semijoins as claimed.

Now let us compare the cost of the join versus the two semijoins. To compute $R_i[A = B]R_j$, one of the relations, R_j say, must be transmitted to the other's site. This has cost $|R_j| * \text{width}(R_j)$, where $\text{width}(R_j)$ equals the number of bits in each tuple of R_j. To compute the semijoins, we transmit $R_j[B]$ to R_i and $R_i[A]$ to R_j, for a cost of $|R_j[B]| * \text{width}(B) + |R_i[A]| * \text{width}(A)$. But $R_i[A] \subseteq R_j[B]$ after $R_i\langle A = B]R_j$ is executed, and so if we execute the semijoins in sequence, the cost is less than or equal to $|R_j[B]| * (\text{width}(A) + \text{width}(B))$. This quantity is less than or equal to $|R_j| * \text{width}(R_j)$ under the reasonable assumption that $\text{width}(A) + \text{width}(B) \le \text{width}(R_j)$. Given this assumption, the cost of the semijoins is less than or equal the cost of the join, as claimed.

Our arguments in support of semijoins are heuristic and there are cases in which joins outperform semijoins. Figure 10 illustrates such a case. An optimal query processing algorithm would almost certainly include both joins and semi-

Let D =	{Ss#,	name,	location),	Y(s#,	p#,	qty),	P(p#,	name,	type)}
	(1,	Acme,	MA	1,	1,	10	1,	LSI,	micro
	2,	Best,	MA	1,	2,	20	2,	P11,	mini
	⋮			⋮			⋮		
	1k,	Mid,	NY				1k,	370,	main
	10k,	Nadir,	CA	1k,	1k,	50	10k,	470,	main

(a) *Attribute domains* dom(S.s#) = dom(Y.s#) = {id#'s from 1 to 10k}
dom(S.name) = dom(P.name)
= {names of length < 10}
dom(S.location) = {states of U.S.}
∨ {provinces of Canada}
etc.

(b) *Auxiliary domains* X_1 = {strings of length < 10}
X_2 = {integers from 1 to 10k}

(c) *Subset hierarchies*

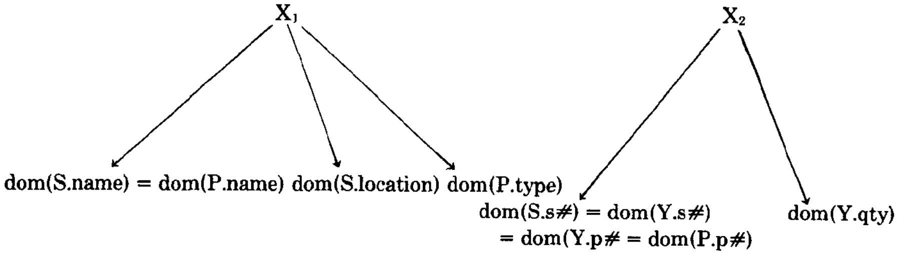


Fig. 11. Domains.

joins. The graceful integration of these tactics is an open problem, however, and our algorithm only uses semijoins.

3. COST AND BENEFIT ESTIMATION

To compile an envelope into an efficient reducer, we need to estimate the cost and benefit of reduction operations. This section presents an estimation procedure based on a statistical model of the database. We only consider the estimation problem for semijoins; estimation techniques for restrictions and projections are described by [28].

Our statistical model is an approximation of a set theoretic model of the database and is described in Section 3.1. Section 3.2 presents a technique for estimating the effect of *set* operations. Section 3.3 extends this technique to estimate the effect of a sequence of semijoins.

3.1 Database Model

Let D = $\{R_1, \ldots, R_n\}$ be a database. Associated with each attribute of each relation, for example, $R_i.A$, is a finite *domain* of values, dom(R_i.A). $R_i[A]$ is constrained to be a subset of dom(R_i.A) (see Figure 11a). The model also contains *auxiliary domains* (see Figure 11b). The set of all domains is partitioned into *domain hierarchies*, each of which contains a maximum domain X_{max} and all domains X such that $X \subseteq X_{max}$ (see Figure 11c). Domains X_i and X_j are *joinable*

domains	X_1	X_2	dom(S.s#)
			dom(Y.s#)
			dom(P.p#)
			dom(Y.p#)
				dom(S.location)

c(domain)	26^{10}	10k	10k	59
w(domain)	10	2	2	2

attributes	S.s#	S.location	P.p#	Y.s#	Y.p# ···
c(attribute)	10k	50	10k	1k	1k

relations	S	Y	P
c(relation)	10k	100k	10k

Fig. 12. Statistical model of database of Figure 12.

if they are members of the same domain hierarchy. If a qualification contains $R_i.A = R_j.B$, then dom(R_i.A) and dom(R_j.B) must be joinable.

We approximate D by the following statistics, called a *database profile*.

(1) For each domain X

  (i) c(A) = the estimated cardinality of X;

  (ii) w(A) = the "width" of X, that is, the number of bits, words, and so forth, used to represent an arbitrary element of X.

(2) For each relation R_i, c(R_i) = the estimated cardinality of R_i.

(3) For each relation R_i and each A $\subseteq$ attr(R_i), c(R_i.A) = the estimated cardinality of R_i[A].

Parameters 1(i) and 1(ii) are fixed a priori by the database administrator, while the other parameters are updated by the system to reflect changes in the database. To reduce overhead, these parameters are updated off-line on a periodic basis.

The statistical model indicates the domain hierarchies by specifying which domains are subsets of which other domains. The model also includes the following assumptions.

(1) If $X_i \subseteq X_j$, then X_i is a *randomly selected* subset of X_j; operationally this means that the probability of $x \in X_i$ is identical for all $x \in X_j$.

(2) For each relation R_i and A $\in$ attr(R_i)

  (i) R_i[A] is a *randomly selected* subset of dom(R_i.A).

  (ii) Tuples of R_i are *uniformly distributed* over values of R_i[A]; this means that the probability of $r_i.A = a$ is identical for all $r_i \in R_i$ and $a \in R_i[A]$.

(3) For each R_i and distinct A, B $\in$ attr(R_i), R_i[A] and R_i[B] are *independent*, meaning that the probability of $r_i.A = a$ is unaffected by the value of r_i.B.

These assumptions are quite strong and this statistical model is a crude approximation. However, it is difficult to devise better models without knowledge of the processes placing data in the database. Figure 12 illustrates our model.

- Let U = X_2 from Figure 11
- Let Σ be the following sequence of operations

Y_1 = select (U, 1)	; Y_1 might represent S[s#]
Y_2 = select (U, 1/10)	; Y_2 might represent Y[s#]
Y_3 = select (Y_1, 1/50)	; Y_3 might represent the effect on S[s#] of S[location = "MA"]
$Y_4 = Y_2 \cap Y_3$	; Y_4 might represent the effect on Y[s#] of Y⟨s# = s#]S.
$Y_5 = Y_3 \cap Y_4$	; Y_5 might represent the effect on S[s#] of S⟨s# = s#]Y

- G(Σ) =

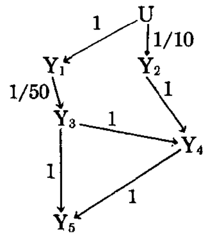


Fig. 13. Graph representations of set operations.

3.2 Effect of Set Operations

Consider the following problem. We are given a *universe* U of objects and two operations for constructing subsets of U—*random selection* (defined below) and *set intersection*. The problem is to estimate the cardinality of any set that can be constructed by a sequence of these operations. Let X be such a set. The *selectivity* of X is the probability that an arbitrary $x \in U$ is also an element of X. The expected $|X|$ is just its selectivity times $|U|$, and so to estimate $|X|$ it is sufficient to estimate its selectivity.

Let $X \subseteq U$ and $x \in U$. We use X(x) as an abbreviation for "$x \in X$," and Prob(X(x)) denotes the probability of X(x) (i.e., the selectivity of X). Similarly, if $S = \{X_1, \ldots, X_m\}$ is a family of subsets of U, S(x) is an abbreviation for $\wedge_{i=1}^{m} X_i(x)$, and Prob(S(x)) denotes the joint probability of x being an element of every $X_i \in S$.

We now define the random selection operation. Let $X \subseteq U$ and $0 \leq \alpha \leq 1$; select(X, α) constructs a set $X' \subseteq X$ in which Prob(X′(x)) = α*Prob(X(x)) for all $x \in U$.

Let Σ be a sequence of selection and intersection operations. We can represent Σ as an edge labeled DAG, G(Σ), whose edges represent operations and whose nodes represent sets constructed by those operations (see Figure 13). Formally, G(Σ) = ⟨V(Σ), E(Σ), label⟩, where

(1) V(Σ) = {U} ∪ S, where S is the family of sets constructed by Σ;
(2) E(Σ) contains the following edges:

(i) ⟨X, X′⟩ with label α if X′ = select(X, α);
(ii) ⟨X, X′⟩ and ⟨Y, X′⟩ with label 1 if $X' = X \cap Y$.

G(Σ) provides an efficient means of calculating Prob(S(x)) for arbitrary families of sets constructed by Σ.

- Let $S = \{U, Y_1, \ldots, Y_5\}$. Lemma 1 states that Prob(S(x)) equals the product of all edge labels that precede S in $G(\Sigma)$.
- *Selectivity* of $U = 1$
  $Y_1 = 1$
  $Y_2 = 1/10$
  $Y_3 = 1/50$
  $Y_4 = 1/500$
  $Y_5 = 1/500$
- Note that $Y_5 = Y_3 \cap Y_4 = Y_3 \cap (Y_2 \cap Y_3) = Y_2 \cap Y_3 = Y_4$, so it is not coincidental that Y_5 and Y_4 have identical selectivities.

Fig. 14. Calculating selectivities for Figure 13.

Lemma 1. *Let Σ be any sequence of selections and intersections operating initially on U, let $S \subseteq V(\Sigma)$, and let $E(S) = \{E \in E(\Sigma) \mid E$ precedes some node $X \in S\}$. Then for all $x \in U$*

$$Prob(S(x)) = \prod_{E \in E(S)} label(E).$$

Proof. See the appendix. □

The lemma is illustrated in Figure 14.

The main result of this section follows a corollary.

Proposition 1. *Let Σ be a sequence of selection and intersection operations operating initially on U, and define $G(\Sigma)$ as above. (i) If $X' =$ select (X, α) is an operation of Σ, then the selectivity of X' equals α times the selectivity of X. (ii) If $X' = X \cap Y$ is an operation of Σ, then the selectivity of X' equals the product over all edges E that precede X or Y in $G(\Sigma)$ of label(E).*

3.3 Effect of Semijoins

A sequence of semijoins is analyzed as several sequences of set operations, one per domain hierarchy. Let Σ_k be the sequence for hierarchy H_k. The universe for Σ_k is the maximum domain of H_k, X_{max}. Σ_k is initialized to contain the following selections.

(1) X = select(X_{max}, α), where $\alpha = c(X)/c(X_{max})$ for each $X \in H_k$.
(2) $R_i[A]$ = select(dom(R_i.A), α) where α = c(R_i.A)/c(dom(R_i.A)) for each relation and attribute such that dom(R_i.A) $\in H_k$.

The effect of a semijoin, $R_i\langle A = B]R_j$ is analyzed in three steps.

(1) The semijoin maps $R_i[A]$ into $R_i[A] \cap R_j[B]$. Suppose dom(R_i.A) $\in H_k$. To estimate the new cardinality of $R_i[A]$, we append $R_i[A] = R_i[A] \cap R_j[B]$ to Σ_k and use Proposition 1 to estimate the new selectivity of $R_i[A]$. c(R_i.A) is updated to the new selectivity times c(X_{max}).

(2) The semijoin eliminates some tuples from R_i. Since tuples of R_i are assumed to be uniformly distributed over values of $R_i[A]$, the estimated new cardinality of R_i is (new value of c(R_i.A))*(old value of c(R_i))/(old value of c(R_i.A)).

(3) The elimination of tuples from R_i causes some values to be deleted from $R_i[A']$, for all $A' \in$ attr(R_i) − {A}. This effect can be analyzed as a *hit ratio* problem: Let t be the old value of c(R_i); these t tuples are assumed to be uniformly

- Consider R(A, B) and suppose | R | = 6 and | R[B] | = 3
- We can partition R into 3 blocks based on B values.

R(A , B)

—	b1
—	b1
—	b2
—	b2
—	b3
—	b3

- If we select one tuple of R we will, of course, hit one block.
- If we select two tuples, we will probably hit two blocks, but we might only hit one.
- If we select three tuples, we might hit three blocks, but it is more likely that we will only hit two.

  And so forth.

Fig. 15. Hit ratio problem.

distributed over b = c(R$_i$.A′) "blocks," where each block contains all tuples with the same A′ value (see Figure 15). The question is: "How many blocks do we expect to hit if we randomly select n = c(R$_i$) tuples?" An efficient formula that answers this question is given by Yao [32]:

the expected number of blocks =

$$Y(n, b, t) = b * \prod_{i=1}^{m} \frac{(td - i + 1)}{t - i + 1}, \qquad \text{where} \quad d = 1 - 1/b.$$

In practice, it is reasonable to approximate Y by

$$\bar{Y}(n, b, t) = \begin{cases} n, & \text{for } n < \tfrac{1}{2}b \\ \tfrac{1}{3}(n + b), & \text{for } \tfrac{1}{2}b < n < 2b \\ b, & \text{for } 2b < n. \end{cases}$$

Y and Ȳ are graphed in Figure 16.

Thus we update c(R$_i$.A′) to Ȳ(new value of c(R$_i$), old value of c(R$_i$.A′), old value of c(R$_i$)).

In addition, we append R$_i$[A′] := select(R$_i$[A′], α) to the sequence for the domain hierarchy that contains dom(R$_i$.A′), where α = (new value of c(R$_i$.A′))/(old value of c(R$_i$.A′)). This selection is not used in estimating the effect of the current semijoin, but is needed to estimate the effects of later ones.

3.4 Cost and Benefit of Semijoins

The *cost* of R$_i$⟨A = B]R$_j$ is defined to be the amount of intersite data transfer required to compute it. This equals

$$\begin{cases} 0, & \text{if } R_i \text{ and } R_j \text{ are stored at the same site} \\ c(R_j.B) * w(dom(R_j.B)), & \text{otherwise.} \end{cases}$$

The *benefit* of R$_i$⟨A = B]R$_j$ is the amount of data eliminated from the database. This equals ((c(R$_i$) before the semijoin) − (c(R$_i$) after the semijoin))∗(the width of R$_i$) = ((c(R$_i$) before) − (c(R$_i$) after))∗ $\Sigma_{A \in attr(R_i)}$ w(dom(R$_i$.A)).

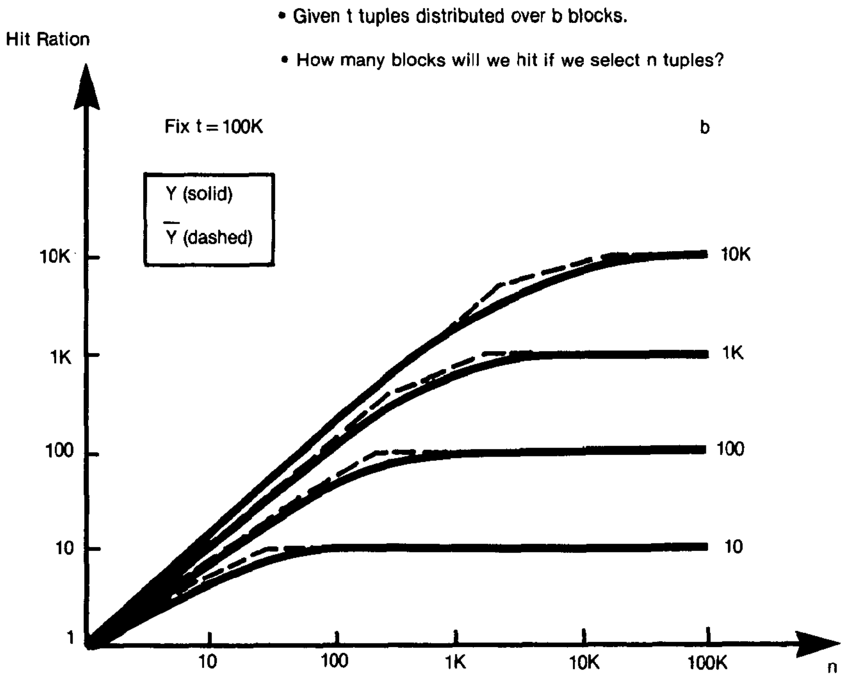


Fig. 16. Yao function.

4. OPTIMIZATION ALGORITHM

This section presents our optimization algorithm. The input is an envelope E and database profile $\bar{D}$. The algorithm compiles E into a reducer P, which is estimated to be profitable in any database modeled by $\bar{D}$. In addition, the algorithm selects an assembly site S_a and appends to P commands to move the reduced database to S_a.

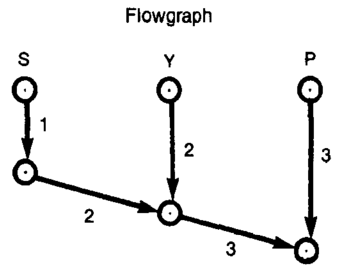
Section 4.1 presents our "basic" algorithm, and Section 4.2 describes two enhancements to the basic algorithm. Section 4.3 illustrates the operation of the algorithm on an example.

4.1 Basic Algorithm

The basic algorithm is an iterative hill-climbing procedure. The algorithm initializes P to contain all *local operations* permitted by E. (Local operations are restrictions, projections, and semijoins whose operands are stored at one site.) The main loop of the algorithm tests whether any nonlocal semijoins permitted by E are profitable. If so, the algorithm selects the most profitable nonlocal semijoin and appends it to P. The algorithms iterates until all profitable semijoins have been exhausted. At this point P is a profitable reducer for E. The algorithm then selects a site S_a and appends commands to move the reduced database to S_a. S_a is selected so as to minimize the quantity of data moved.

Reducer

1. S[location = "MA"]
2. Y<s# = s#]S
3. P<p# = p#]Y



- The first node in each column represents the initial state of the relation.
- Edge (1) represents operation (1).
- Each semijoin is represented by two edges, one vertical and one diagonal.
- Diagonal edges represent data flow between relations, while vertical edges represent successive reductions of a single relation.

Fig. 17. Flowgraphs for reducers.

The basic algorithm is listed as follows.

```
ALGORITHM OPT
   Input: envelope E and profile $\bar{D}$.
 Output: reducer P augmented by commands to move reduced database to $S_a$.
1.  Initialization
1.1   P := sequence of all local operations permitted by E
1.2   Estimate the cost and benefit of all nonlocal semijoins permitted by E
2.  Main Loop
2.1   Do while some nonlocal semijoin permitted by E has benefit > cost
2.2      Let sj be the most profitable nonlocal semijoin permitted by E
2.3      Append sj to P
2.4      Estimate the effect of sj and update costs and benefits accordingly
2.5   end
3.  Termination
3.1   For each site S, let size(S) = the sum of $c(R_i)*w(R_i)$ over all relations $R_i$ referenced
      by E and stored at S
3.2   Select $S_a$ to be the site with maximum size
3.3   Append to P commands to move data from all other sites to $S_a$
end
```

4.2 Enhancements

Algorithm OPT is an example of a *greedy* optimization algorithm; it always seeks to maximize *immediate* gain, it never looks ahead, and never backs up. As a result, the reducers generated by OPT are, in general, suboptimal. This section presents two techniques for improving OPT. These enhancements help compensate for OPT's greed by considering the indirect effects of semijoins.

These enhancements are most easily described in terms of data flowgraphs [18]. Let P be a reducer. Its flowgraph G(P) is a directed graph whose nodes represent the intermediate results of P and whose edges represent the operations of P (see Figure 17). By convention we draw flowgraphs in columns, each of which contains nodes that represent reductions of one relation. With this convention, the edges of the flowgraph are partitioned into *vertical edges* and *diagonal edges.*

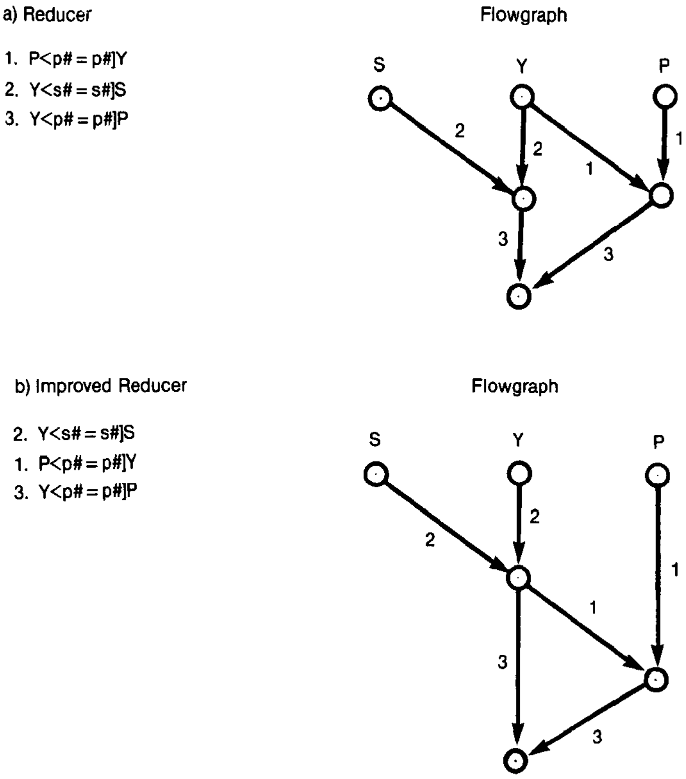


Fig. 18. Enhancements.

Each semijoin is represented by one vertical and one diagonal edge. Intuitively, the diagonal edge "carries the cost" of the semijoin; for example, the cost of $R_i \langle A = B]R_j$ equals the size of $R_j[B]$ (assuming R_i and R_j are at different sites), and the diagonal edge represents this data flow.

The first enhancement permutes the order of semijoins in P to decrease the cost of some semijoins without increasing the cost of any others. Consider Figure 18a. Semijoin 1 (P⟨p# = p#]Y) uses Y to reduce relation P, and semijoin 2 (Y⟨s# = s#]S) reduces Y. Since semijoin 2 reduces Y, the cost of semijoin 1 can be decreased by delaying it until semijoin 2 executes (see Figure 18b). This permutation also increases the effect of semijoin 1, since semijoin 2 increases the selectivity of Y[s#], and so the cost of each subsequent semijoin is decreased as well. Thus this permutation is guaranteed to lower the cost and increase the benefit of the reducer.

This transformation can be visualized as reducing the cost of a semijoin by delaying its diagonal edge. For example, in Figure 18, we have reduced the cost of semijoin 1 by delaying its diagonal edge until semijoin 2 has been executed. More generally, let P be any reducer, let $\langle N_j, N_i \rangle$ be any diagonal edge in G(P), and let N_j' be any node that follows N_j in the same column. The replacement of $\langle N_j, N_i \rangle$ by $\langle N_j', N_i \rangle$ monotonically decreases the cost of P, provided the resulting graph remains acyclic. (If the resulting graph is cyclic, it no longer represents a sequential program.)

Let P be the output of OPT. We apply the above transformation to the semijoins of P considered in decreasing cost order. Thus we delay the most expensive semijoins in P to take advantage of reductions achieved by other semijoins.

Our second enhancement prunes semijoins from P that are rendered unprofitable by the choice of assembly site S_a. Consider Figure 18a and suppose Y's site is selected to be S_a. This choice renders semijoin 3 (Y⟨p# = p#]P) useless, and this semijoin should be discarded. With respect to semijoin 2 (Y⟨s# = s#]S), the situation is less clear-cut. Although there is no direct benefit in reducing the size of Y, semijoin 2 is indirectly beneficial via semijoin 1 (P⟨p# = p#]Y). In fact, semijoin 2 both decreases the cost and increases the benefit of semijoin 1.

In general, let P be the output of Algorithm OPT; that is, P is a reducer augmented by commands to move the reduced database to S_a. For each relation R_i stored at S_a, and for each semijoin in P of the form $R_i\langle A = B]R_j$ we compare the cost of P to the cost of P without the semijoin. If the latter cost is lower, the semijoin is pruned from P.

4.3 Example

In this section we simulate the optimization procedure on the following envelope and profile.

Database S(s#, name, location), Y(s#, p#), P(p#, name, type)

Profile	X_1 = dom(S.s#) = dom(Y.s#) ;	$c(X_1) = 100{,}000$
	X_2 = dom(P.p#) = dom(Y.p#) ;	$c(X_2) = 100{,}000$
	X_3 = dom(S.location) ;	$c(X_3) = 50$
	X_4 = dom(P.type) ;	$c(X_4) = 5$

All domains have widths of 1.

	S	(s#,	name,	location)
est. cardinality:	10000	(10000,	na,	50)
	Y	(s#,	p#)	
est. card.:	100000	(1000,	1000)	
	P	(p#,	name,	type)
est. card.:	10000	(10000,	na,	5)

Each relation is stored at a separate site.

Envelope

Retrieve into S(s#, name, location) where **qual**
Retrieve into Y(s#, p#) where **qual**
Retrieve into P(p#, name, type) where **qual**
qual: S. location = "MA" ∧ P.type = "micro"
∧ S.s# = Y.s# ∧ Y.p# = P.p#

4.3.1 *Initialization.* The local operations permitted by the envelope are

S[location = "MA"]
P[type = 'micro']

The estimated effect of these operations is to reduce the size of S by a factor of 50 and P by a factor of 5 (see Figure 19a).

Flowgraphs are augmented here to show the estimated cardinality of relations and "important" projections. Numbers in the S (resp. P) column indicate c(S) and c(S.s#) (resp. c(P) and c(P.p#)). Entries in the Y column have the form c(Y)
c(Y.s#), c(Y.p#)

Reducer

a) S[location = 'MA']
P[type = 'micro']

b) Y<s# = s#]S

c) P<p# = p#]Y

d) Y<p# = p#]P

e) S<s# = s#]Y

f) Move S to Y's site
Move P to Y's site

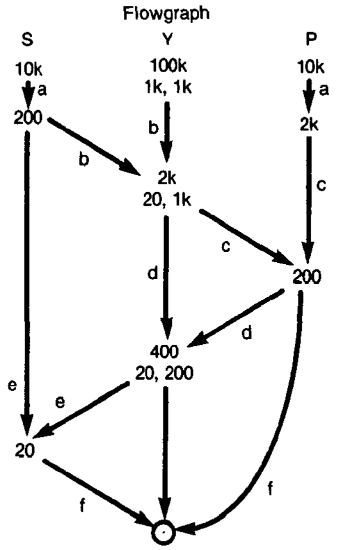


Fig. 19. Example reducer.

4.3.2 *Main Loop.* The estimated cost and benefit of each permitted semijoin is listed as follows.

	Semijoin	Cost	Effect	Benefit
1.	S⟨s# = s#]Y	1000	c(S.s#) = c(S) = 20	180 * 3
2.	P⟨p# = p#]Y	1000	c(P.p#) = c(P) = 200	1800 * 3
3.	Y⟨s# = s#]S	200	c(Y.s#) = 20 c(Y) = 2k c(Y.p#) = 1k	98k * 2
4.	Y⟨p# = p#]P	2000	c(Y.p#) = 200 c(Y) = 20k c(S.s#) = 1k	80k * 2

Semijoin 3 is most profitable and would be selected. The estimated effect of this semijoin is summarized in Figure 19b. Costs and benefits of semijoins are updated as follows.

	Semijoin	Cost	Effect	Benefit
1.	S⟨s# = s#]Y	20	c(S.s#) = c(S) = 20	180 * 3
2.	P⟨p# = p#]Y	1000	c(P.p#) = c(P) = 200	1800 * 3
3.	Y⟨s# = s#]S	200	none	none
4.	Y⟨p# = p#]P	2000	c(Y.p#) = 200 c(Y) = 400 c(Y.s#) = 20	1600 * 2

The most profitable semijoin is 2 and the estimated effect is summarized in Figure 19c. Costs and benefits are updated as follows.

	Semijoin	Cost	Effect	Benefit
1.	S⟨s# = s#]Y	20	c(S.s#) = c(S) = 20	180 * 3
2.	P⟨p# = p#]Y	1000	none	none
3.	Y⟨s# = s#]S	200	none	none
4.	Y⟨p# = p#]P	200	c(Y.p#) = 200 c(Y) = 400 c(Y.s#) = 20	1600 * 2

Semijoin 4 is most profitable and has the effect shown in Figure 19d. Costs and benefits are updated once again.

Reducer

a) S[location = 'MA']
P[type = 'micro']

b) Y<s# = s#]S

c) P<p# = p#]Y

e) S<s# = s#]Y

f) Move S to Y's site
Move P to Y's site

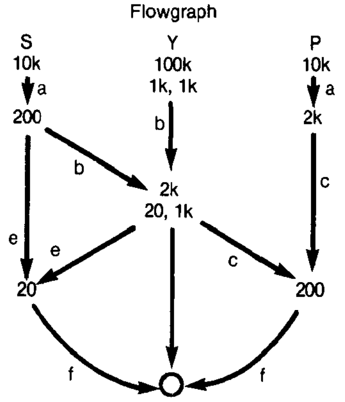


Fig. 20. Improved reducer for Figure 19.

	Semijoin	Cost	Effect	Benefit
1.	S⟨s# = s#]Y	20	c(S.s#) = c(S) = 20	180 * 3
2.	P⟨p# = p#]Y	200	none	none
3.	Y⟨s# = s#]S	200	none	none
4.	Y⟨p# = p#]P	200	none	none

The only profitable semijoin is the first one. Its effect is indicated in Figure 19e. No further semijoins are profitable, and so the main loop terminates.

4.3.3 *Termination.* Y is estimated to be the largest relation in the reduced database, and so Y's site is selected as S_a. The final program produced by Algorithm OPT constitutes Figure 19a–f. The estimated cost of this program is

```
  1420  (for semijoins—see Figure 19b–e)
+  660  (for assembly—see Figure 19f)
  2080  (total)
```

4.3.4 *Enhancements.* The first enhancement in Section 4.2 does not apply to this example, but the second enhancement does. In particular, semijoin 4 (Y⟨p# = p#]P; see Figure 19d) should be pruned from the reducer. The resulting program has cost 1880 and appears in Figure 20.

5. DISCUSSION

We have presented an algorithm for processing queries in a distributed database. In this section we discuss the relationships between our algorithm and other work in this area; we also suggest topics for future research.

The first comprehensive algorithm for distributed query processing algorithms was developed by Wong [30]. Wong's algorithm translates a query Q into a sequence of two tactics: (1) *move* a subrelation from one site to another; and (2) *process* data at one site using relational operations. The algorithm is a recursive optimization procedure. It begins by selecting a site S_a and then constructing the following initial solution.

(1) Move all relations referenced by Q to S_a.
(2) Process Q at S_a as a local query, using the relations moved in step 1.

The initial solution is improved by recursively replacing individual "move" commands by lower cost sequences of "move" and "process" commands. The algorithms terminates when no "move" command can be replaced by a lower cost sequence. This algorithm produces increasingly efficient sequences of commands, although its hill-climbing discipline is too weak to guarantee optimality.

Wong's algorithm was implemented in SDD-1 and early experience indicated problems with the algorithm's a priori selection of S_a [22]. To mitigate this problem, Rothnie developed a branch-and-bound technique that permits parallel consideration of several assembly sites.

Our algorithm is a further refinement of Wong's algorithm, in which the concepts of semijoin and reducer are used to abstract the main optimization problem. These concepts simplify the algorithm and provide a framework for future research. These concepts also improve the effectiveness of the algorithm by (1) avoiding the need for a priori selection of assembly site, (2) suggesting enhancements to the basic algorithm, and (3) supporting mathematically sound cost estimation techniques.

Other distributed query processing algorithms are described by [7, 8, 10, 13, 16, 17, 19, 27]. Hevner and Yao [16, 17] consider a class of relational queries (*simple queries*) that have the form

$$\text{Retrieve } R_1(A), \text{ where } R_1.A = R_2.A \wedge R_2.A = R_3.A \wedge \cdots \wedge R_{n-1}.A = R_n.A.$$

Observe that this query is computing $R_1[A] \cap R_2[A] \cap \cdots \cap R_n[A]$, and semijoins are powerful enough to solve such queries. Hevner and Yao present an efficient algorithm that constructs an optimal sequence of semijoins for solving a given simple query.[5] They also extend their algorithm to produce efficient (not necessarily optimal) reducers for relational queries that are approximately as general as envelopes.

Chiu [7] considers queries (*chain queries*) of the form

$$\text{Retrieve } R_1(A_1),$$
$$\text{where } R_1.A_1 = R_2.A_1 \wedge R_2.A_2 = R_3.A_2 \wedge \cdots \wedge R_{n-1}.A_{n-1} = R_n.A_{n-1}.$$

(Chain queries are so named because their join graphs are chains.) Semijoins are powerful enough to solve chain queries, and Chiu presents an efficient dynamic programming algorithm that constructs an optimal sequence of semijoins for solving a given chain query. Chiu and Ho [8] generalize Chiu's approach and develop a methodology for producing optimal semijoin programs for queries (*tree queries*) whose join graphs are trees.

Distributed query processing algorithms that use joins instead of semijoins are proposed by [10, 27]. An algorithm that uses joins *and* semijoins is described by [19]; this algorithms, however, only considers "star" networks.

Finally, we observe that semijoins have been recommended in disguised forms in nondistributed query processing contexts. Several associative memory database machines provide hardware semijoin instructions in lieu of join. Examples include CAFS [1], CASSM [29], and RAP [21].[6] To apply our algorithm to these machines, it is only necessary to modify our cost estimation techniques. Semijoin

[5] Semijoin is called "join-project" in [17].

[6] Semijoin is called "join using bit array" in CAFS, "match" in CASSM, and "implicit join" in RAP.

is also fundamental to the INGRES query processing algorithm [31], where it is called "detachment." Theoretical research on semijoins includes [2-4, 12, 33, 34].

The conceptual simplicity of our optimization algorithm suggests areas requiring additional research.

1. A major problem is the hill-climbing discipline of Algorithm OPT. The algorithm selects semijoins that maximize immediate gain, ignoring the fact that the execution of one semijoin often decreases the cost and increases the benefit of other semijoins. A principal topic for future research is to develop efficient optimization algorithms that exploit this fact. The work of [7, 8, 16, 17] represents preliminary steps in this direction.

2. The programs constructed by our optimization procedure consist of a *reduction phase* followed by a *final processing phase*. The reduction phase executes all cost-effective semijoins permitted by the query in a distributed fashion, while the final processing phase executes all joins needed to solve the query in a centralized fashion. However, it is sometimes better to execute joins in a distributed fashion also, and the execution of some joins may render additional semijoins profitable. This suggests an alternate query processing strategy with the following structure.

```
Do while the query is not solved
        (i)   execute all profitable semijoins
        (ii)  execute one or more joins
end
```

Evidently the critical new optimization problem is to select the joins that are processed on each iteration.

3. All query processing algorithms developed to date are "open-loop" and cannot respond to errors in cost estimation. To close the loop, we must be able to halt execution of a reducer in midstream and construct a new reducer that utilizes (a) the partial results already computed and (b) the cost information obtained by the partial computation.

4. Fundamental to our approach is the assumption that intersite data transfer is the dominant cost of distributed query processing. Our basic strategy of reducing the database before processing the query makes sense because of this assumption. In systems where this assumption does not hold, our techniques may still apply, provided it is cheaper to compute reductions than to solve queries.

APPENDIX. PROOF OF LEMMA 1

The proof is by induction on the length of Σ.

Basis Step. $|\Sigma| = 0$. In this case, $G(\Sigma)$ is the singleton graph containing U and the result is immediate.

Induction Step. Assume the lemma holds when $|\Sigma| < L$; prove that it still holds when $|\Sigma| = L$.

Let Σ' comprise the first $L-1$ operations of Σ, and let X' be the Lth set constructed by Σ. $G(\Sigma')$ is a subgraph of $G(\Sigma)$. $G(\Sigma)$ contains exactly one node that is not in $G(\Sigma')$—namely, the node representing X'— and either one or two edges not in $G(\Sigma')$—namely, the edges corresponding to the Lth operation of Σ. These nodes and edges cannot precede any node or edge in $G(\Sigma')$ by construction.

Let S′ be any subset of the nodes of Σ′. By induction hypothesis

$$\text{Prob}(S'(x)) = \prod_{\text{E that precede S}' \text{ in G}(\Sigma')} \text{label(E)}$$

$$= \prod_{\text{E that precede S}' \text{ in G}(\Sigma)} \text{label(E), by argument above.}$$

Thus the lemma holds for all sets of nodes that do not include X′. Let S = S′ ∪ {X′}. There are two cases depending on whether the Lth operation is a selection or an intersection.

Case 1. X′ = select (X, α). By definition of S(x),

$$\begin{aligned}\text{Prob}(S(x)) &= \text{Prob}(S'(x) \wedge X'(x)) \\ &= \text{Prob}(S'(x) \wedge X(x) \wedge X'(x)) \qquad \text{since} \quad X'(x) => X(x) \\ &= \alpha * \text{Prob}(S'(x) \wedge X(x)), \qquad \text{by definition of select.}\end{aligned}$$

Observe that S′ ∪ {X} are nodes of G(Σ′), and so by induction hypothesis

$$\text{Prob}(S'(x) \wedge X(x)) = \prod_{\text{E that precede S} \cup \{X\} \text{ in G}(\Sigma')} \text{label(E)}.$$

Now, the edges that precede S = S′ ∪ {X′} in G(Σ) are identical to those that precede S′ ∪ {X} in G(Σ′) plus the edge ⟨X, X′⟩ that represents the operation X′ = select(X, α); the latter edge has weight α. Thus

$$\prod_{E \in E(S)} \text{label(E)} = \alpha * \prod_{\text{E that precede S}' \cup \{X\} \text{ in G}(\Sigma')} \text{label(E)}$$

$$= \alpha * \text{Prob}(S'(x) \wedge X(x)), \quad \text{as desired.}$$

Case 2. X′ = X ∩ Y. By definition of S(x),

$$\begin{aligned}\text{Prob}(S(x)) &= \text{Prob}(S'(x) \wedge X'(x)) \\ &= \text{Prob}(S'(x) \wedge X(x) \wedge Y(x)) \quad \text{by definition of intersection.}\end{aligned}$$

Observe that S′ ∪ {X, Y} are nodes of G(Σ′), and so by hypothesis

$$\text{Prob}(S'(x) \wedge X(x) \wedge Y(x)) = \prod_{\text{E that precede S}' \cup \{X,Y\} \text{ in G}(\Sigma')} \text{label(E)}.$$

The edges that precede S = S′ ∪ {X′} in G(Σ) are identical to those that precede s′ ∪ {X, Y} in G(Σ)′ plus the edges ⟨X, X′⟩ and ⟨Y, X′⟩ that represent the operation X′ = X ∩ Y; the latter edges have weight 1. Thus

$$\prod_{E \in E(S)} \text{label(E)} = \prod_{\text{E that precede S}' \cup \{X,Y\} \text{ in G}(\Sigma')} \text{label(E)}$$

$$= \text{Prob}(S'(x) \wedge X(x) \wedge Y(x)), \quad \text{as desired.} \qquad \square$$

REFERENCES

1. Babb, E. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst. 4*, 1 (March 1979), 1–29.

2. Bernstein, P.A., and Chiu, D.W. Using semijoins to solve relational queries. *J. ACM. 28*, 1 (Jan. 1981), 25–40.

3. BERNSTEIN, P.A., AND GOODMAN, N. The power of natural semijoins. *SIAM J. Comput. 10*, 4 (Nov. 1981).
4. BERNSTEIN, P.A., AND GOODMAN, N. The power of inequality semi-joins. To appear in *Inf. Syst.*
5. BERNSTEIN, P.A., AND SHIPMAN, D.W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 52–68.
6. BERNSTEIN, P.A., SHIPMAN, D.W., AND ROTHNIE, JR., J.B. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 18–51.
7. CHIU, D.M. Optimal query interpretation for distributed databases. Ph.D. Dissertation, Harvard Univ., Cambridge, Mass., Dec. 1979.
8. CHUI, D.M., AND HO, Y.C. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proc. ACM-SICMOD Conf.*, May 1980.
9. DATE, C.J. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 1977.
10. EPSTEIN, R., STONEBRAKER, M., AND WONG, E. Distributed query processing in a relational database system. In *Proc. ACM-SIGMOD Conf.*, June 1978, pp. 169–180.
11. GOODMAN, N., BERNSTEIN, P.A., WONG, W., REEVE, C. L., AND ROTHNIE, JR., J.B. Query processing in SDD-1. Tech. Rep. CCA-79-06, Computer Corp. of America, Cambridge, Mass., Oct. 1979.
12. GOODMAN, N., AND SHMUELI, O. Tree queries: A simple class of relational queries. To appear in *ACM Trans. Database Syst.*
13. GROUDA, M.G., AND DAYAL, U. Optimal semijoin schedules for query processing in local distributed database systems. In *Proc. ACM-SIGMOD Conf.*, April 1981, pp. 164–175.
14. HAMMER, M.M., AND SHIPMAN, D.W. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM Trans. Database Syst. 5*, 4 (Dec. 1980), 431–466.
15. HELD, G.D., STONEBRAKER, M., AND WONG, W. INGRES—A relational database management system. In *Proc. AFIPS 1975 NCC*, vol. 44, AFIPS Press, Arlington, Va., pp. 409–416.
16. HEVNER, A.R. Optimization of query processing in distributed databases. Ph.D. Dissertation, Purdue Univ., Lafayette, Ind., Sept. 1979.
17. HEVNER, A.R., AND YAO, S.B. Query processing in distributed database systems. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 177–187.
18. KARP, R.M., AND MILLER, R.E. Properties of a model for parallel computation: Determinary, termination, queueing. *SIAM J. Appl. Math. 14* (Nov. 1966), 1390–1411.
19. KERSCHBERG, L., TING, P.D., AND YAO, S.B. Query optimization in star computer networks. Unpublished Rep., Bell Laboratories, Holmdel, N.J., 1980.
20. MARILL, T., AND STERN, D.H. The datacomputer: A network data utility. In *Proc. AFIPS 1975 NCC*, vol. 44, AFIPS Press, Arlington, Va.
21. OZKARAHAN, E.A., SCHUSTER, S.A., AND SEVCIK, K.C. Performance evaluation of a relational associative processor. *ACM Trans Database Syst. 2*, 2 (June 1977), 175–195.
22. ROTHNIE, J.B. Private communication. See also: A distributed database management system for command and control applications: Semiannual technical report 2. Tech. Rep. CCA-80-03, Computer Corp. of America, Cambridge, Mass., Jan. 1980.
23. ROTHNIE, JR., J.B., BERNSTEIN, P.A., FOX, S.A., GOODMAN, N., HAMMER, M.M., LANDERS, T.A., REEVE, C.L., SHIPMAN, D.W., AND WONG, E. A system for distributed databases (SDD-1). *ACM Trans. Database Syst. 5*, 1 (March 1980), 1–17.
24. ROTHNIE, J.B., AND GOODMAN, N. An overview of the preliminary design of SDD-1. In *Proc. Berkeley Workshop Distributed Data Management and Computer Newtorks*, May 1977, pp. 39–57.
25. ROTHNIE, J.B., AND GOODMAN, N. A survey of research and develoment in distributed database systems. In *Proc. 3rd Int. Conf. Very Large Databases*, Oct. 1977, pp. 48–61.
26. ROTHNIE, J.B., GOODMAN, N., AND MARILL, T. Database architecture in a network environment. In *Protocols and Techniques for Data Communication Networks*, F.F. Kuo, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1980.
27. SELINGER, P.G., AND ADIBA, M. Access path selection in distributed database management systems. In *Proc. Int. Conf. Databases*, Univ. Aberdeen, Aberdeen, Scotland, July 1980.
28. SELINGER, P.G., ASTRAHAN, M.M., CHAMBERLIN, D.D., LORIE, R.A., AND PRICE, T.G. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Conf.*, June 1979, pp. 23–34.

29. Su, S.Y.W., and Emam, A. CASDAL: CASSM's data language. *ACM Trans. Database Syst. 3*, 1 (March 1978), 57-91.

30. Wong, E. Retrieving dispersed data from SDD-1. In *Proc. Berkeley Workshop Distributed Data Management and Computer Networks*, May 1977, pp. 217-235.

31. Wong, E., and Youssefi, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223-241.

32. Yao, S.B. Approximating block accesses in database organizations. *Commun. ACM 20*, 4 (Apr. 1977), 260-261.

33. Yu, C.T., and Ozsoyoglu, M.Z. An algorithm for tree-query membership of a distributed query. In *Proc. Compsac79*, IEEE Computer Society, Nov. 1979, pp. 306-312.

34. Yu, C.T., and Ozsoyoglu, M.Z. On determining tree query membership of a distributed query. Tech. Rep. TR80-1, Dep. Computing Science, Univ. Alberta, Edmonton, Alta., Canada, Jan. 1980.

Received October 1979; revised June 1980, accepted December 1980