0-315-05535-7

53429

National Library of Canada | Bibliothèque nationale du Canada

**CANADIAN THESES ON MICROFICHE** | ***THÈSES CANADIENNES SUR MICROFICHE***

NAME OF AUTHOR/*NOM DE L'AUTEUR* PATRICK ALLEN BLACK

TITLE OF THESIS/*TITRE DE LA THÈSE* QUERY PROCESSING ON DISTRIBUTED DATABASE SYSTEMS

UNIVERSITY/*UNIVERSITÉ* SIMON FRASER UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/ *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* MASTER OF SCIENCE

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE* 1982

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* DR. WO-SHUN LUK

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

*L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.*

*L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.*

DATED/*DATÉ* APRIL 14, 1982 SIGNED/*SIGNÉ*

PERMANENT ADDRESS/*RÉSIDENCE FIXE*

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Ottawa, Canada
K1A 0N4

NL-339 (Rev. 8/80)

# QUERY PROCESSING ON DISTRIBUTED DATABASE SYSTEMS

by

Patrick A. Black

B.Sc., University of British Columbia, 1972

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

© Patrick A. Black 1982

SIMON FRASER UNIVERSITY

January, 1982

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

APPROVAL

Name: Patrick Allen Black

Degree: Master of Science

Title of Thesis: Query Processing on Distributed Database Systems

Examining Committee:

Chairperson: Thomas K. Poiker

Wo Shun Luk
Senior Supervisor

Nick Cercone

Richard F. Hobson

Brian Alspach
External Examiner
Professor
Department of Mathematics
Simon Fraser University

Date Approved: January 7, 1982

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

QUERY PROCESSING ON DISTRIBUTED DATABASE SYSTEMS

Author:

(signature)

PATRICK ALLEN BLACK

(name)

APRIL 14, 1982

(date)

ABSTRACT

Query processing on a distributed database system requires the transmission of data between computers on a communication network. Minimizing the amount of data transmission is important to reduce query processing costs and to prevent network congestion.

The semi-join operation is important in formulating query processing strategies. It provides significant reductions in the amount of data communication required in processing queries. The accurate estimation of database state reductions by semi-join operations is necessary. Current distributed database system models are inadequate in this respect. A new distributed database system model is developed to this end and is utilized in this research. The method for determining correct semi-join cost/benefit is shown.

We have developed algorithm BLACK which contains heuristics for generating semi-join preprocessing strategies for queries.

We have constructed a simulation model to evaluate the performance of existing query preprocessing algorithms (algorithms AP, OPT and HEVNER) and algorithm BLACK. This model tests random queries and the results are presented and discussed. It is shown that the performance of algorithms AP, OPT and HEVNER produce semi-join preprocessing strategies that perform poorly in comparison to algorithm BLACK. A 'speed-up' algorithm (WS) is utilized to further improve upon algorithm BLACK's performance.

# ACKNOWLEDGEMENTS

I wish to thank my senior supervisor Wo-Shun Luk for his invaluable guidance of my research, and Nick Cercone, Rick Hobson and Max Krause for proofreading my thesis and making valuable comments. I wish to express my gratitude to Ethel Inglis for her invaluable help.

Table of Contents

Page

APPROVAL .......... ii

ABSTRACT .......... iii

ACKNOWLEDGEMENTS .......... iv

LIST OF TABLES .......... viii

LIST OF FIGURES .......... ix

Chapter

1. DISTRIBUTED DATABASE SYSTEMS .......... 1
   1.1 INTRODUCTION .......... 1
   1.2 MOTIVATION OF THESIS .......... 3
   1.3 PLAN OF THESIS .......... 5
2. THE RELATIONAL DATA MODEL .......... 7
   2.1 INTRODUCTION AND DEFINITIONS .......... 7
   2.2 RELATIONAL OPERATIONS .......... 9
   2.3 RELATIONAL QUERIES .......... 12
   2.4 ALTERNATIVE AND OTHER RELATIONAL DATABASE MODELS .......... 13
3. DISTRIBUTED QUERY PROCESSING .......... 15
   3.1 INTRODUCTION .......... 15
   3.2 DATA TRANSMISSION COST FUNCTION .......... 17
   3.3 ATTRIBUTE VALUE ASSUMPTIONS .......... 19
   3.4 COST/BENEFIT OF A SEMI-JOIN .......... 20
   3.5 COMPLEXITY CONSIDERATIONS .......... 22
   3.6 DATABASE STATE UPDATE CONSIDERATIONS .......... 23
4. SEMI-JOIN COST/BENEFIT ESTIMATION .......... 24

4.1 INTRODUCTION ....................................24

4.2 THE EFFECT OF SEMI-JOINS ON RELATION AND JOINING ATTRIBUTE CARDINALITY ....................................26

4.2.1 QUERIES WITH A SINGLE JOIN ATTRIBUTE ............30

4.2.2 QUERIES WITH MULTIPLE JOIN ATTRIBUTES ...........33

4.3 THE EFFECT OF SEMI-JOINS ON NON-JOINING ATTRIBUTE CARDINALITIES ....................................38

4.4 BENEFIT AND UPDATE CALCULATION ALGORITHMS .............41

4.5 IMPLICATIONS FOR A DISTRIBUTED DATABASE SYSTEM MODEL ..45

5. PREPROCESSING STRATEGIES FOR DISTRIBUTED QUERIES ..........47

5.1 INTRODUCTION ....................................47

5.2 THE INITIAL FEASIBLE SOLUTION ....................48

5.3 ALGORITHM HEVNER ....................................48

5.4 ALGORITHM AP ....................................54

5.5 ALGORITHM BLACK ....................................56

5.6 ALGORITHM WS ....................................59

5.7 ORDER OF COMPLEXITY OF THE ALGORITHMS ................60

5.8 AN EXAMPLE ....................................60

5.8.1 HEVNER'S RESULT ....................................61

5.8.2 AP'S RESULT ....................................62

5.8.3 BLACK'S RESULT ....................................62

6. SIMULATION MODEL AND RESULTS ............................64

6.1 INTRODUCTION ....................................64

6.2 GENERATING RANDOM QUERIES ............................64

6.2.1 VARYING THE NUMBER OF JOIN CLAUSES ..............65

6.2.1.1 QUERIES WITH A MINIMUM NUMBER OF JOIN CLAUSES ....................................66

6.2.1.2 QUERIES WITH A MAXIMUM NUMBER OF JOIN CLAUSES ....................................67

6.2.1.3 QUERIES WITH AN INTERMEDIATE NUMBER OF JOIN CLAUSES ....................67

6.2.2 VARYING THE NUMBER OF RELATIONS, ATTRIBUTES AND AVERAGE SELECTIVITY ....................67

6.3 RANDOM DATABASE DESCRIPTIONS ....................68

6.4 STEADY STATE CONSIDERATIONS ....................68

6.5 RESULTS ....................68

6.5.1 IMPORTANCE OF THE INITIAL FEASIBLE SOLUTION ....................69

6.5.2 COST COMPARISON OF THE IFS TO SEMI-JOIN PREPROCESSING STRATEGIES ....................70

6.5.3 IMPORTANCE OF THE NUMBER OF SEMI-JOINS ....................73

6.5.4 SEMI-JOIN PROGRAM COST ....................76

6.5.5 COMPARISON OF ALGORITHMS ....................79

6.5.5.1 RESULTS FROM VARYING THE NUMBER OF RELATIONS ....................79

6.5.5.2 RESULTS FROM VARYING THE NUMBER OF ATTRIBUTES ....................83

6.5.5.3 RESULTS FROM VARYING THE AVERAGE SELECTIVITY ....................87

6.5.5.4 SUMMARY OF RESULTS ....................90

7. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK ....................91

REFERENCES ....................93

LIST OF TABLES


TABLE TITLE PAGE

1 Average AVG_COST, varying No. of Relations...........82

2 Average AVG_COST, varying No. of Attributes..........86

3 Average AVG_COST, varying Average Selectivity........87

LIST OF FIGURES


| FIGURE | TITLE | PAGE |
| --- | --- | --- |
| 1 | Lemma 2, graph G | 34 |
| 2 | Lemma 2, graph G' | 34 |
| 3 | Lemma 3, graph G | 37 |
| 4 | Lemma 3, graph G' | 38 |
| 5 | IFS COST/AVG_COST versus No. of Relations | 71 |
| 6 | IFS COST/AVG_COST versus No. of Attributes | 71 |
| 7 | IFS COST/AVG_COST versus Average Selectivity | 72 |
| 8 | No. of Semi-Joins versus No. of Relations | 75 |
| 9 | SJ COST/AVG_COST versus No. of Relations | 77 |
| 10 | SJ COST/AVG_COST versus No. of Attributes | 77 |
| 11 | SJ COST/AVG_COST versus Average Selectivity | 78 |
| 12 | AVG_COST versus No. of Relations for a Minimum No. of Join Clauses | 80 |
| 13 | AVG_COST versus No. of Relations for an Intermediate No. of Join Clauses | 80 |
| 14 | AVG_COST versus No. of Relations for a Maximum No. of Join Clauses | 81 |
| 15 | AVG_COST versus No. of Attributes for a Minimum No. of Join Clauses | 84 |
| 16 | AVG_COST versus No. of Attributes for an Intermediate No. of Join Clauses | 84 |
| 17 | AVG_COST versus No. of Attributes for a Maximum No. of Join Clauses | 85 |
| 18 | AVG_COST versus Average Selectivity for a Minimum | |

No. of Join Clauses........................................88

19 AVG_COST versus Average Selectivity for an
Intermediate No. of Join Clauses....................88

20 AVG_COST versus Average Selectivity for a Maximum
No. of Join Clauses........................................89

# 1. DISTRIBUTED DATABASE SYSTEMS

## 1.1 INTRODUCTION

Typically in a 'computerized information system' we have a database and methods to access data from the database in response to queries. These database systems usually exist at an organizations central office, in private enterprise or in a government agency or department. There is also a need for geographically distributed components of these organizations to access data from central databases.

One method to satisfy this need is to establish a communication link between the central computer and the site where data are required. This method has its drawbacks for several reasons. If telecommunication lines are utilized, either a dedicated communication line or the public communication lines may be utilized. If a dedicated communication link exists, the cost of this link is often exhorbitant. If the public communication lines are used, one must contend with problems. Lines are required to establish a link with the central computer which may be unavailable if lines are busy. In any case there is always the dial up time required to establish the link with the central computer. The central computer may also be serving the maximum number of users and service may not be available. Another drawback is system reliability; if the central computer malfunctions, data access is denied. As the information needs of the organization grow it becomes more difficult and more expensive to increase the speed and capacity of the central

computer. It is foreseeable that a single central computer operating with the latest technology may not be able to operate efficiently on a large database.

Another method of satisfying this need for distributed data is to provide each geographically dispersed site with a complete copy of the central database. One problem with this approach is the simultaneous updating of data at each site so that a consistent version of the database is available to all users. This approach would also involve expensive hardware costs.

A popular intermediate approach, which is receiving much attention in the literature [Good79, Hevn79a, Hevn79b, Ston76, Wong79, Yu79], is to have data distributed such that the data most necessary to, and most often accessed by, a site is present at that site. In this manner the large central database is divided into smaller databases and most data would be distributed non-redundantly among the sites, although some data may be replicated. The site computer and database system need not be as large as the single centralized version. System reliability is enhanced since the failure of a site computer generally only affects that site. Hardware and software updates are much simpler and less costly as local demands increase. Communication between sites is only required when the data required by a query is not resident at the site of origin. This attractive intermediate approach solves information processing needs of geographically dispersed organizations. We call such a system a distributed database system.

There are many unsolved problems with distributed database

systems. Research is currently being done in the areas of distributed concurrency control, system reliability, failure and recovery, distributed database design, distributed system control, query processing and security. The focus of this research is on distributed database design with particular emphasis on the query processing aspect. The motivation for this research is discussed in more detail in the next section.

## 1.2 MOTIVATION OF THESIS

The need for communication between sites distinguishes a distributed database system from a centralized database system. A low level view of this communication is digitized data being transmitted from site to site. The purpose of this communication is to access data at other sites in the network and for transmission of data in response to queries.

Queries in database systems are typically expressed in terms of application programs written in high level procedural languages for batch processing of queries. Interactive query processing is especially popular in distributed systems. High level query languages such as INGRES' QUEL [Ston76] have been utilized to this end. An example of a QUEL query on a supplier database is shown below.

```
RETRIEVE (SUPPLIER.PART_NUM)
  WHERE SUPPLIER.PART_NUM = ORDER.PART_NUM AND
        ORDER.PRICE > 1000
```

This query requests all part numbers in the SUPPLIER relation such that the part numbers are on order (in the ORDER relation)

and these part numbers have their parts price greater than $1000. High level query languages permit complex data extraction without the necessity of procedurally specifying how it is to be extracted. The relational data model is an ideal data model for this approach. Codd [Codd70] defined the relational data model and showed how data may be accessed in terms of relational algebra and relational calculus, both of which form the basis for high level query languages. A high level query may be interpreted by the local distributed database management system as commands (operations) that will extract the necessary distributed data to satisfy queries.

An aspect of query processing on distributed database systems that has received much attention is what operations (relational ones) will be utilized in processing queries. The semi-join operation [Bern81a, Bern81b, Chiu80, Hevn79b] (half a join in relational algebra) has been found to be a good strategy for query processing on distributed systems. Semi-join may be utilized to reduce relation sizes of distributed relations. Since semi-join may be defined in mathematical terms which form the basis for relational set theory and since its physical operations on data are clearly defined, semi-join is a very useful tactic for processing queries on distributed database systems.

To answer queries, one strategy is to move all relevant relations to the node where the result is required. This is a costly data communication process. Semi-join is used to reduce the size of distributed relations before they are transmitted on

the network to the node where the result of the query is desired. Considerable research [Bern81a, Bern81b, Chiu80, Hevn79a, Hevn79b, Wong79] has been done concerning semi-join tactics and their use in distributed database systems. Three algorithms [Good79, Hevn79a, Wong79] have been published that produce semi-join strategies for preprocessing queries. In order to formulate a query processing strategy one must assume a particular model of the distributed database system. This model should correctly estimate the effect of semi-join on the database state. One motivaton of this work is to review the existing models of distributed database systems. A goal of this work is to define a comprehensive model of a distributed database system, since the models presently in use are shown to be inadequate.

## 1.3 PLAN OF THESIS

The issues discussed above are investigated in later chapters. The relational data model is described in the context of a distributed database system. The relational operations that we will utilize are introduced and the applicability of the relational data model and other database models to this area is discussed.

In Chapter 3 we discuss query processing on distributed database systems. The basic assumptions of our work in this area are given. Our views on communication network data transmission and incurred costs are presented. Semi-join is introduced and its use as a query preprocessing strategy is outlined. The

complexity of determining optimal query preprocessing strategies is described. Finally, we introduce the concept of database state updates (i.e. how is the database affected by a semi-join operation).

In Chapter 4 we investigate the problems discussed in the last section of Chapter 3. The correct cost/benefit determination of a semi-join is formulated. We introduce the semi-join 'history' problem and determine a solution for it. We develop estimating functions to determine the resultant cardinalities of a relation (and its attributes) when it is reduced by a semi-join. We introduce an algorithm, which combines these results, that allows us to make correct updates of the database state. In the last section, we discuss the implications and applications of the results presented in this chapter.

In Chapter 5, we extend the discussion of using semi-join as a preprocessing strategy for queries. We introduce two published algorithms [Good79, Hevn79a] which produce semi-join preprocessing strategies. We introduce an algorithm (BLACK) we have developed and discuss its approach. Finally an example is given to demonstrate each of these algorithms.

In Chapter 6, we introduce a simulation model that tests the performance of the algorithms mentioned above. We discuss how to generate random queries and describe the classes of random queries that were tested in the simulation program. In the last section we analyze and discuss the results of the simulation.

# 2. THE RELATIONAL DATA MODEL

## 2.1 INTRODUCTION AND DEFINITIONS

The relational data model was originally formulated by Codd [Codd70, Codd71, Codd72]. The relational data model provides a formal, high-level description of a collection of data items (i.e. a database) in terms of relations. The terms domain, attribute, relation, relation schema, relation state, database schema and database state, contribute to the definition of the relational data model. These terms are defined as follows:

1. Domain A set of data values.
2. Attribute A name given to a set of data values. The set of data values an attribute name describes is a subset of the possible values in a specific domain of values. Hence, each attribute is uniquely defined on one domain of values.
3. Relation A relation may be thought of as a table in which each row of the table is called a tuple and each column of the table is labelled by the attributes comprising the relation. No two tuples in the relation are identical.
4. Relation Schema A relation schema names the relation and the attributes in that relation.
5. Relation State The contents of a relation at some moment in time is called the relation state.
6. Database Schema A set of relation schemas.
7. Database State A set of relation states such that there is one relation state per relation schema.

In this work, relation names are capital letters from the latter part of the alphabet (e.g. Q, R) or the capital letter, R, indexed by the integers (e.g. R1, R2 etc.). Attribute names are either capital letters from the first part of the alphabet (e.g. A, B) or the capital letter, A, indexed by the integers (e.g. A1, A2 etc.). Some specific relation examples will use more descriptive relation and attribute names.

Given the preceding definitions, we can define the following relation and attribute parameters.

For a relation R, let:

n = number of tuples (the relation cardinality).

a = number of attributes.

sR = size of R, $sR = n * \prod_{i=1}^{a} Wi$

For an attribute Ai, let:

Di = number of possible domain values (the domain cardinality).
Ci = number of distinct values currently in Ai (the attribute cardinality).
Wi = size of a data item in Ai.

sAi = projected size of the attribute with no duplicate values, $sAi = Wi * Ci$.

We assume Wi, the size of a data item in Ai, is one unit of data in size. This assumption does not limit the generality of our work since the projected size of an attribute is always directly proportional to the cardinality of the attribute and the size of a relation is always directly proportional to the cardinality of that relation. Furthermore this assumption allows us to make the following simplifications on the relation and attribute parameters.

For a relation R, then:

n = number of tuples.

a = number of attributes.

sR = size of R, sR = n * a.

For an attribute Ai, then:

Di = number of possible domain values.

Ci = number of distinct values currently in Ai.

sAi = projected size of the attribute with no duplicate values, sAi = Ci.

In the next section the relational operations will be introduced.

## 2.2 RELATIONAL OPERATIONS

Each node in a distributed database contains a local database. Each local database is described by a database schema and a particular manifestation of this database schema, the database state. Relational operations in particular the join operation may be defined intra-nodally (within a node) or inter-nodally (between nodes).

We now introduce the relational operations.

1. Restriction A restriction is an intra-nodal operation and may be represented as follows: R[A=x]. Attribute A in relation R is to be restricted to those values in column A such that they equal the value x. (In practise, any appropriate boolean condition may be applied rather than equals.) Formally:

$$R[A=x] = \{r \in R \mid r.A=x\}$$

where r.A is the value of attribute A in tuple r.

2. Projection A projection is an intra-nodal operation and may be represented as follows: R[A]. The projection of relation R on attribute A is obtained by eliminating all columns of R not labelled by A and then eliminating duplicate tuples. Formally:

$$R[A] = \{r.A \mid r \in R\}$$

This definition may be easily generalized to projection on a set of attributes from relation R.

3. Join A join operation may be represented as follows: R[A=B]S. The join operation is used to combine two relations. Attributes A and B must be defined on the same domain. A value of A in R is compared with a value of B in S. If the two values have the relationship specified in the join operation (e.g. '=', equality), then the tuples of the relations are combined to form a third relation. Formally:

$$R[A=B]S = \{rs \mid r \in R,\ s \in S, \text{ and } r.A=s.B\}$$

The join is inter-nodal if the two relations are at different nodes in the network. To perform an inter-nodal join, one of the relations must be moved in its entirety to the node where the other relation resides.

4. Semi-join The semi-join is used as an inter-nodal operation and may be represented as follows: R<A=B]S. The semi-join operation may be used to perform the join of two relations without moving an entire relation to perform the join. The semi-join is performed by first projecting S.B at the node where relation S resides. S.B is transmitted to the node

where relation R resides. R is then restricted to those values of R.A equal to those values in S.B. If the reverse semi-join were then performed (i.e. S<B=A]R', where R' is the result of the first semi-join) we would effectively have completed the join of relations R and S. Formally:

$$R<A=B]S = \{r \in R \mid (\exists s \in S)(r.A=s.B)\}$$

The following example illustrates the above operations. Given the two relations R1 and R2, with relation states:

R1:

| A | B |
|---|---|
| 1 | unary |
| 3 | ternary |
| 2 | binary |

R2:

| C | D |
|---|---|
| - | 1 |
| - | 2 |
| * | 2 |
| / | 2 |
| + | 2 |

The restriction, R2[D=2], has the following result.

R2':

| C | D |
|---|---|
| - | 2 |
| * | 2 |
| / | 2 |
| + | 2 |

The projection, R2[C], has the following result.

R2':

| C |
|---|
| - |
| * |
| / |
| + |

The intra or inter-nodal join, R1[A=D]R2, has the following result.

R3:

| A | B | C | D |
|---|---|---|---|
| 1 | unary | - | 1 |
| 2 | binary | - | 2 |
| 2 | binary | * | 2 |
| 2 | binary | / | 2 |
| 2 | binary | + | 2 |

The semi-join, R1<A=D]R2, has the following result at the node at which R1 resides.

R1':

| A | B |
|---|---|
| 1 | unary |
| 2 | binary |

If we wish to complete the join of R1 and R2, we first perform the reverse semi-join, R2<D=A]R1', with the result at the node at which R2 resides.

R2':

| C | D |
|---|---|
| - | 1 |
| - | 2 |
| * | 2 |
| / | 2 |
| + | 2 |

and then, if R1' and R2' at some point are at a common node, a join of R1' and R2' on attributes A and D will produce the same result as the join R1[A=D]R2.

## 2.3 RELATIONAL QUERIES

Queries in a relational database system may be expressed using high-level non-procedural query languages. Codd's original paper [Codd70] set the basis for two families of relational query languages, relational calculus and relational algebra. The definitions of restriction, projection, join and semi-join presented earlier are expressed in relational algebra. We assume a relational algebra interface to our distributed database system.

Queries are then formed of two parts, a 'target list' and a 'qualification' part. The target list is a list of the attributes to be output in the result of the query. The

qualification part is a boolean combination of restriction clauses and join clauses. The attributes in the target list are implied projections. We assume that qualifications are pure conjunctions. Disjunction is handled by placing the qualification in disjunctive normal form and treating each conjunction separately. Join clauses are all assumed to be equi-join clauses (i.e. R[A=A]S) for simplicity of treatment. The distributed aspect of relational queries will be discussed in Chapter 3.

## 2.4 ALTERNATIVE AND OTHER RELATIONAL DATABASE MODELS

Many database models are in use today. For example there is the DBTG model [CODA71] from the CODASYL group, IBM's hierarchical data model, IMS [IBM78], and a relational database model, SYSTEM R [Astr76]. The nodes in a distributed database system may utilize different database models. It is necessary that the distributed database system operate under a single database model to simplify translations between local database models and to unify the query processing aspect. Given a particular database schema under a particular database model, a set of relations may be defined which represents a translation in terms of the relational data model [Zani79]. The relational data model is an ideal data model for this translation because of its generality. From the front-end point of view, relational queries may be expressed in high-level non-procedural type query languages. Other database models operate on a 'one record at a time' basis and are not amenable to this approach since data

communication between nodes tends to be costly. In this sense other database models are considered to be 'low-level' and the relatonal data model is considered to be 'high-level'. Thus the results described in this work are applicable regardless of the underlying internal database structures.

# 3. DISTRIBUTED QUERY PROCESSING

## 3.1 INTRODUCTION

The objective of query processing on distributed database systems is to have users express queries as if the distributed database were a single unified database. The fact that data is actually distributed physically is transparent to the user. The user should however be able to direct the result of a query to any node in the network. This node is termed the 'result node'.

Queries are analyzed by the local distributed database management system. Since data may be stored redundantly at different nodes in the distributed database system, it is necessary for the local distributed database management system to determine a non-redundant manifestation of the distributed database to utilize in answering a query. For example, if a query references a part number relation for automotive parts and there is a copy of this relation at two different nodes in the network, then which manifestation of this relation is utilized in answering the query? The problem of determining which manifestation to use is not within the scope of this research. Thus we henceforth assume that data are stored non-redundantly.

If the query has no distributed data requirements then it may be solved using local processing alone. Otherwise the query requires distributed processing. At each of the set of nodes where data is required, we assume that there is a single relation from which to access data. If two or more relations at a node are referenced in a query, we can join these relations

using an intra-nodal join to form a single unified relation.

Once a query has been expressed we need to follow some query processing scheme to answer the query. The query processing scheme in general use [Good79, Hevn79a] is as follows:

1. Initial local processing. The relational operations of projection, restriction and intra-nodal join are done first to reduce the amount of data before any data transmissions are made.
2. Processing strategy. A sequence of data transmission steps and local processing steps are done to further reduce the amount of data involved in answering the query. This step can be considered to be a preprocessing step (for the next step).
3. Final data transmissions. Data is then transmitted from the distributed nodes to the result node where final local processing is done to form the result of the query.

Steps 1 and 3 represent well known techniques in database management systems. Step 2 has been the focus of much research in recent years [Good79, Hevn79a, Hevn79b, Wong79]. Without processing strategies, whole relations would always have to be transmitted on the network to answer queries. This would lead to exhorbitant data communication costs and a high likelihood of a severely congested communication network. This research investigates query processing strategies.

## 3.2 DATA TRANSMISSION COST FUNCTION

We devote our attention to the distributed query processing aspect and we assume that all initial local processing (Step 1) has already been performed, that is, projections, restrictions or intra-nodal joins have already been performed. We are left with a query referencing N distributed nodes at each of which one single relation exists. The query consists of a set of equi-join clauses, on any number of attributes.

The costs in processing this type of query are:

1. Local processing cost. We assume, as other workers have done [Bern81b, Hevn79a], that any further local processing required has zero cost. This assumption is justified since data transmission on a network is known to be costly vis a vis local processing.
2. Data transmission cost. Hevner and Yao [Hevn79b] have assumed that for each data transmission done on the network there exists a start-up cost in addition to the cost of transmitting the data. This cost function is expressed in the following formula:

   $$DT(X) = C + X$$

   where $DT(X)$ is the cost of transmitting X amount of data on the network and C is the start-up data transmission cost constant. This cost function assumes distance has no effect on the cost of a data transmission. In the future this may be a valid assumption. This function also assumes that X amount of data can be transmitted in a single transmission. This certainly is possible but considering the recent

proliferation of packet switching technology [Mart76], it is unlikely that this would be the case. We therefore do not assume this cost function but assume the data transmission cost to be directly proportional to the amount of data transmitted. This assumption is valid if the amount of data to be transmitted is greater than the size of a packet. Packets in packet switched networks appear to average about 1024 bits in size [Mart76]. If PS is the packet size then our data transmission cost function may be expressed as follows:

$$DT(X) = CEILING(X \text{ MODULO } PS) * C + X$$

If X is always greater than PS then DT(X) is directly proportional to X. We can then ignore the first term above and quote results relative to the amount of data transmitted, X.

The literature is divided on which cost function to use. SDD-1 [Good79] uses the cost function we have adopted.

It is necessary to define in what units, the amount of data transmitted on the network, X, will be measured. Our attribute values have been defined to be one unit of data in size, perhaps a byte or a word. It is not important to define exactly the number of bits of data transmitted on the network. We are concerned only with the relative amounts of data being transmitted. Typically, we will be transmitting on the network a column (i.e. the distinct values of a particular attribute) from a given relation. The cost of this data transmission is defined to be the number of units of data in that column.

For example, if we have an attribute A in R1, then C(R1.A) is the cardinality of that attribute. C(R1.A) is also the size of that attribute (in units of data as we have defined them). If C(R1.A) is 1000 and R1[A] is transmitted on the network, then the cost of this data transmission is 1000. We do not consider attribute value sizes greater than one unit of data. Clearly, in a typical database, attribute value sizes vary. We have made this assumption for simplicity in the treatment of attribute and relation sizes and it does not affect the generality of the results presented in this research.

## 3.3 ATTRIBUTE VALUE ASSUMPTIONS

It is necessary to make some assumption about how the attribute values are distributed in an attribute. If we know this distribution then we can statistically estimate the result of, say, a semi-join on that attribute. (This estimating problem is tackled in the next chapter.)

Assumption 1: In each attribute, the discrete values are uniformly distributed.

The probability that a tuple has a particular value is the same as the probability that it has any other value. This is clearly a critical assumption since it will be utilized in forming query processing strategies. If attribute value distributions are non-uniformly distributed then the strategies formed will not perform as predicted. This assumption is quite strong and this statistical model is a crude approximation. For example, a salary attribute would be unlikely to have uniformly

distributed data values. Each attribute in the database is likely to have its own characteristic distribution of data values. These distributions may also change over time. To model this situation is indeed complex. Better statistical models may be devised and utilized when the particular characteristics of a given database are known.

Within a relation, an attribute may be either dependent or independent with respect to other attributes in that relation. If some dependency does exist then the problem of estimating cardinality reductions of the attributes becomes very complex.

Assumption 2: Each attribute in a relation is independent of all other attributes in the same relation.

This assumption will allow us to estimate cardinality changes of all attributes in a relation.

Assumption 3 is similar in nature to Assumption 2 but it deals with attribute independence between relations.

Assumption 3: An attribute in a relation is independent of any attributes in other relations.

The three assumptions are necessary for formulating query processing strategies since these strategies will need to make estimates of how relation and attribute cardinalities will be affected by relational operations.

## 3.4 COST/BENEFIT OF A SEMI-JOIN

After all initial processing is complete, the only operations left to perform are joins between the distributed relations (inter-nodal joins). Since these joins require moving

whole relations on the network, we use the semi-join operation as a preprocessing tactic. The rationale for the use of semi-join tactics is described next.

Let R and Q be relations at different nodes in the network and say we have the join R[A=A]Q to perform. Let the sizes of R and Q be 2000 and the cardinaltiy of R.A and Q.A be 500 and 1000, respectively. If we transmit one data item on the network, we say the cost in data communication is one (unit of data). If the join were performed on the network it would require transmitting 2000 units of data on the network. To perform this join using semi-joins we first execute the semi-join Q<A=A]R. This is done by first projecting on attribute A in relation R, R[A], then transmitting R[A] to relation Q, requiring the transmission of 500 units of data on the network. Relation Q is joined with R[A] to get relation Q'. Assume Q'.A has a resultant cardinality of 500. (We discuss how to make this estimate rigorously in the next chapter.) The semi-join R<A=A]Q' is then performed by first projecting on attribute A in relation Q' and transmitting Q'[A] on the network to R, requiring the transmission of 500 units of data on the network. Q'[A] is then joined with R to get relation R'. Effectively, R and Q have now been joined. The cost of this join is conservatively, the transmission of 1000 units of data on the network. The above tactics are illustrated below.

JOIN TACTIC:
Move R to site S.
Transmission Cost = 2000 data items

SEMI-JOIN TACTIC:
Q<A=A]R (Move R[A] to site S)
Transmission Cost = 500 data items
R<A=A]Q' (Move Q'[A] to site R)
Transmission Cost = 500 data items

Total Transmission Cost = 1000 data items

The 'benefit' of a semi-join is the amount of data reduction at the relation being reduced by the semi-join. A semi-join is said to be cost beneficial if the cost of data transmission incurred by that semi-join is less than or equal to the benefit. The semi-join operation is then used as a preprocessing tactic to reduce the cardinality of relations distributed on the network before they are transmitted to the result node.

Our query processing strategies will utilize the semi-join operation exclusively. We will call the sequence of semi-joins that perform these data reductions a 'semi-join program'. The semi-join program represents a 'preprocessing strategy' for the query. A semi-join program is intended to be executed sequentially; i.e. the first semi-join in the program is executed first, the second semi-join is executed second, etc.

## 3.5 COMPLEXITY CONSIDERATIONS

Since, in the general query environment, the problem of determining an optimal semi-join preprocessing strategy is an NP-HARD problem [Hevn79a], algorithms which generate these preprocessing strategies are necessarily heuristic in nature.

## 3.6 DATABASE STATE UPDATE CONSIDERATIONS

A problem is encountered when semi-join is used as a query preprocessing tactic. When a particular semi-join program is generated, the cost beneficial semi-joins are added to it incrementally. For each semi-join added to the semi-join program, the database state needs updating to reflect the execution of this semi-join so that the next semi-join to be considered may have its correct cost and benefit determined. We have found that the 'history' of previous semi-joins that are already in the semi-join program can affect the effect on the database state of a new semi-join that is being considered for addition to the semi-join program. This history problem is dealt with in the next chapter of this work. The solution to this problem leads to a more accurate definition of a distributed database system model.

# 4. SEMI-JOIN COST/BENEFIT ESTIMATION

## 4.1 INTRODUCTION

The main concern of this chapter is the cost/benefit estimation of a candidate semi-join (abbreviated CSJ) for possible addition to the semi-join program (to be referred to as RHO). The cost of the CSJ depends on the current state of the database. The benefit, which is the amount of reduction that could be obtained after the semi-join is performed, will involve a hypothetical update of the database state. Thus the cost/benefit estimation problem is reduced to the problem of estimating the database state each time a semi-join has been added to RHO, i.e. an update problem.

The current state of the database consists of the cardinality (i.e. the number of tuples) of each relation and the cardinality of each attribute (i.e. the number of distinct values) in each relation. Given a new semi-join to be added to RHO, we only update the state of the relation that is being reduced. A simple updating method, used in [Yu79], [Hevn79a] and [Good79], is as follows: (define this calculation as the normal effect calculation (abbreviated NEC))

Consider the semi-join Rj<A=A]Ri, where Rj is being reduced. Then

$$\text{(NEC)} \quad \begin{aligned} CC(Rj) &:= CC(Rj) * CC(Ri.A)/DC.A \\ CC(Rj.A) &:= CC(Rj.A) * CC(Ri.A)/DC.A \end{aligned}$$

where CC(Rj) and CC(Rj.A) are the current cardinalities (abbreviated CC) of Rj and Rj.A respectively and DC.A stands for

the domain cardinality of attribute A. Note that the factor CC(Ri.A)/DC.A is crucial in the calculation. It is sometimes referred to in the literature as the selectivity.

What is often ignored in the literature is the fact that the validity of NEC depends on two important assumptions:

1. attribute Rj.A is independent of attribute Ri.A (Assumption 3 in Chapter 3.)

2. Rj.A is independent of Rj.B, B being any attribute in the relation other than A. (Assumption 2 in Chapter 3.)

Assumption 2 is often taken for granted in the literature. With this assumption we can isolate semi-joins with one attribute from semi-joins with other join attributes and consider the effects of these semi-joins within this subset. Though Assumption 1 is of a similar nature, it does have a different implication here. This is because within the subset of relations which are affected by the semi-joins with the same attribute, say A, the estimates of CC(Rj.A) and CC(Ri.A) may not be derived independently of each other. EXAMPLE 1, introduced below, illustrates this point.

Consider the distributed database which has three relations, R1, R2, and R3 with 10, 50, and 1000 tuples respectively and attribute A in R1, R2, and R3 with 10, 45, and 50 distinct values respectively (domain cardinality of A is 50). Let the query be such that it requires join operations, R2[A=A]R1 and R3[A=A]R1.

<u>EXAMPLE 1:</u>

```
  Semi-join 1:     R2<A=A]R1
  Semi-join 2:     R3<A=A]R2
  Semi-join 3:     R3<A=A]R1
```

In Semi-join 1, R2.A is restricted to those tuples of R1.A that join with it. In Semi-join 2, R3.A is restricted to those tuples of R2.A that join with it, after Semi-join 1. At this point R3[A] $\subseteq$ R1[A]. The execution of Semi-join 3 will result in no benefit (reduction) at R3 but we will acquire the cost of Semi-join 3 in this query preprocessing strategy. Worse, if there were semi-joins after this one, the cost and benefit of each of them would be estimated based on an incorrect database state.

Other anomalies may arise when the history of previous semi-joins is ignored in the cost/benefit estimation. This chapter corrects these anomalies.

The next section describes how to determine the effect of a semi-join on the cardinality of the relation being reduced and on the cardinality of the joining attribute in the relation being reduced. The next section describes how to estimate the effect of a semi-join on the non-joining attributes in the relation being reduced. The next section summarizes these results in the form of algorithm UPDATE. The last section of this chapter describes how to use the results presented in this chapter.

## 4.2 THE EFFECT OF SEMI-JOINS ON RELATION AND JOINING ATTRIBUTE CARDINALITY

In this section, we determine how to estimate the effect of a semi-join, say R<A=A]S, on the cardinality of relation R and

on the cardinality of the joining attribute, R.A. First, we introduce some definitions that are necessary in the discussion that follows.

A distributed relational database system consists of N relational database systems at N sites. Portions of the database are stored nonredundantly at each site. We denote a query by Q, with qualification, C. Local operations such as restrictions, projections, and joins of relations in the same node are assumed to have no cost and are excluded from C. Thus each site is assumed to contain only one relation. C is then a conjunction of clauses of the form Rj[A=A]Ri, where Rj and Ri are relations stored at sites j and i, respectively. The closure of C, denoted C+, includes all clauses implied by C under transitivity [Good79]. Let J+ be the set of semi-joins implied by C+ (e.g. if Rj[A=A]Ri is in C+ then Rj<A=A]Ri and Ri<A=A]Rj are in J+).

Algorithms that generate semi-join programs will utilize a subset of J+ in generating a semi-join program (to be referred to as RHO) that is to be executed on the distributed database. We assume that once a semi-join is added to RHO it is not considered again as a possible addition to RHO. Each semi-join in RHO is represented by the double: k,Rj<A=A]Ri where k indicates semi-join Rj<A=A]Ri is the kth semi-join added to RHO and is the kth semi-join that will be executed on the database. This semi-join is often called SJ-k where k is some positive integer.

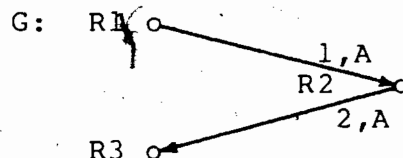
We use a graph representation for a semi-join program. Let G be the graph of the semi-join program RHO generated by an

algorithm so far. As each new semi-join is added to RHO, G is updated as follows: For the kth semi-join, ( k, Rj<A=A]Ri ), if Ri (or Rj) is not a vertex in G then add a new vertex Ri (or Rj) to G. Add a new directed edge from Ri to Rj in G labelled with k,A. An A_PATH in G from Rm to Rn is defined to be a path of directed edges labelled with the same attribute from Rm to Rn in G such that the k component of the edge labels from Rm to Rn be in strictly increasing order. If Rj is on an A_PATH in G from Ri then Ri is called an A_PREDECESSOR of Rj.

This graph, G, is used to trace the past history of CC(Rj.A) for every relation, Rj, and join attribute A. In EXAMPLE 1 there is already an A_PATH from R1 to R3 (i.e. R1 is an A_PREDECESSOR of R3) so that R1[A] and R3[A] are already related. This explains why NEC does not work for EXAMPLE 1. We shall now introduce two other circumstances under which NEC does not work.

EXAMPLE 2:

RHO:
  1,R2<A=A]R1
  2,R3<A=A]R2




and CSJ: R1<A=A]R3

In G, after SJ-2, R1 is an A_PREDECESSOR of R3. The CSJ makes R3 an A_PREDECESSOR of R1. (This is in contrast to EXAMPLE 1 where the third semi-join makes R1 redundantly an A_PREDECESSOR of R3.) Since after SJ-2, R3[A] $\subseteq$ R1[A], the effect of the CSJ on R1 will be:

CC(R1) := CC(R1) * (CC(R3.A)/CC(R1.A))

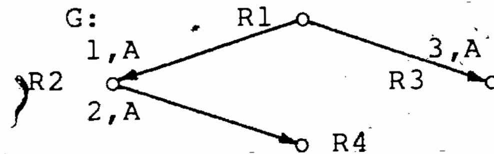
and

CC(R1.A) := CC(R3.A)

Comparing this result with NEC, we note that the selectivity of the CSJ in effect here is CC(R3.A)/CC(R1.A), where CC(R1.A) is the cardinality of R1.A before the semi-join.

EXAMPLE 3:

RHO:
1,R2<A=A]R1
2,R4<A=A]R2
3,R3<A=A]R1




and CSJ: R4<A=A]R3

After SJ-3, R3 and R4 are on parallel A_PATHs in G and R1 is an A_PREDECESSOR of both R3 and R4. Since R4[A] $\subseteq$ R1[A] and R3[A] $\subseteq$ R1[A], R3[A] and R4[A] are related. Thus the CSJ should not have the reducing effect on R4 suggested by NEC. Knowing the history of semi-joins which have an effect on the relations in the CSJ is important. This problem is dealt with in the next section.

These examples provide a small insight into the problem of accurate effect estimation. It shows that when the two relations involved in the CSJ have some common A_PREDECESSORs they are no longer independent; thus, the assumption under which NEC is valid is no longer true. We conclude this section with the following lemma.

LEMMA 1: The normal effect calculation, NEC, of CSJ, Rj<A=A]Ri, works correctly if and only if the set of all A_PREDECESSORs of Rj and the set of all A_PREDECESSORs of Ri are disjoint.

PROOF: If the two sets do not intersect, then the two values, CC(Rj.A) and CC(Ri.A) are derived independently of each other. By the assumptions we have made earlier, NEC applies.

If the two sets intersect, then one of the following cases must exist:

(1) Ri is an A_PREDECESSOR of Rj. Hence Rj[A] $\subseteq$ Ri[A] and Ri.A and Rj.A are not independent.

(2) Rj is an A_PREDECESSOR of Ri. Hence Ri[A] $\subseteq$ Rj[A] and Rj.A and Ri.A are not independent.

(3) There exists some Rk that is an A_PREDECESSOR of both Ri and Rj but neither (1) or (2) is true. Hence Ri[A] $\subseteq$ Rk[A] and Rj[A] $\subseteq$ Rk[A] and Ri.A and Rj.A are not independent.

Thus NEC is correct only if the two sets are disjoint.

### 4.2.1 QUERIES WITH A SINGLE JOIN ATTRIBUTE

In this section we solve the semi-join history problem for query qualifications involving only a single join attribute. The single attribute solution will provide an insight for a more rigorous treatment in the next section for the general case where we shall consider queries with more than one join attribute.

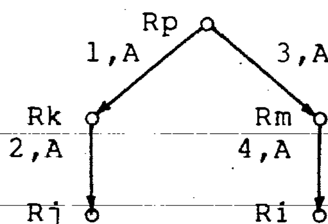
Consider the following example where all semi-joins have the same join attribute:

EXAMPLE 4:

RHO:

1,Rk<A=A]Rp
2,Rj<A=A]Rk
3,Rm<A=A]Rp
4,Ri<A=A]Rm

G:



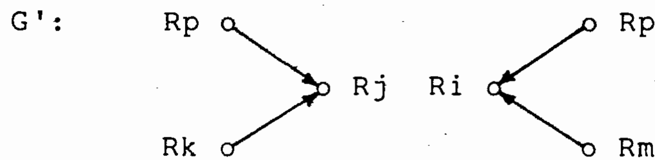

and CSJ: Rj<A=A]Ri

Consider the states of Rj.A and Ri.A before the CSJ is applied. By (NEC) we have:

```
 CC(Rj.A)  = OC(Rj.A)  * OC(Rp.A)/DC.A
                       * OC(Rk.A)/DC.A
and
 CC(Ri.A)  = OC(Ri.A)  * OC(Rp.A)/DC.A
                       * OC(Rm.A)/DC.A
```

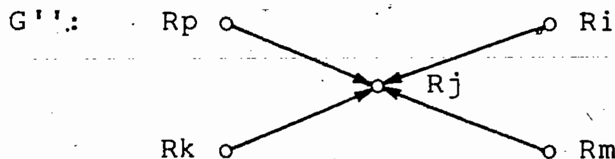
where OC is the abbreviation for original cardinality. We see that the effect of the semi-join SJ-1 on Rj.A (via Rk) is equivalent to adding its selectivity as a factor in the calculation of CC(Rj.A). The order of application of the semi-joins SJ-1 and SJ-2 is unimportant. Hence G is equivalent to G' below:

G':




After the CSJ is applied on Rj, G' would become G'' as follows:

G'':




where the duplicate of Rp is removed from G'.

We now introduce a data structure, A_LIST, which has the capability to represent G' or G''. For attribute A with relations {R1,R2,...,Rm} in the qualification, define its A_LIST structure initially to be:

```
R1 :   R1
R2 :   R2
  ...
Rm :   Rm
```

The relations on the left are indices and the list on the right contains those relations by which the index relation on the left

has been reduced. As each new semi-join, say Rj<A=A]Ri, is added to RHO, A_LIST(Rj) is replaced by the union of A_LIST(Ri) and A_LIST(Rj). That is, all relations on an A_PATH to Ri are added to A_LIST(Rj) as relations on an A_PATH to Rj.

The A_LIST state of EXAMPLE 4 is:

```
Rp :   Rp
Rk :   Rk, Rp
Rj :   Rj, Rk, Rp
Rm :   Rm, Rp
Ri :   Ri, Rm, Rp
```

If we add the CSJ to RHO then after the update, A_LIST(Rj) will contain Rp, Rm, Ri, Rj and Rk and:

$$CC(Rj.A) = OC(Rj.A) * OC(Rp.A)/DC.A * OC(Rm.A)/DC.A * OC(Rk.A)/DC.A * OC(Ri.A)/DC.A$$

More generally, if there are r elements in A_LIST(Rj), i.e. {R1,...,Rr}, (Note Rj is always an element of A_LIST(Rj))) then the current cardinality of Rj.A is given by:

$$CC(Rj.A) = OC(Rj.A) * \prod_{k=1,\ k \neq j}^{r} OC(Rk.A)/DC.A$$

The update method is as described above. If the previous and current cardinalities of Rj.A are C1 and C2 respectively, then the effective selectivity of the semi-join Rj<A=A]Ri is defined to be C2/C1. If the previous cardinality of the relation Rj is C3, then the current cardinality is C3*(C2/C1). If Rj contains 'a' attributes, then the benefit of the semi-join is (C3*a)-((C3*C2/C1)*a), given the assumption made on attribute value size in Chapter 2. The cost is the cardinality of Ri.A.

### 4.2.2 QUERIES WITH MULTIPLE JOIN ATTRIBUTES

We now extend our result to query qualifications containing more than a single join attribute.

Consider the case where semi-joins on different attributes are considered. If the set of attributes referenced in J+ is the set {A1,A2,...,Am} we have the following A_LIST structures to consider:

A1_LIST, A2_LIST, ... , Am_LIST

When we considered the single attribute case it was sufficient to know the original cardinality (OC) of each relation that had reduced, say, Rj. However, if Rk has reduced Rj on attribute A, but Rk itself has been reduced earlier by Rp on attribute B, then it is C(Rk.A) (the cardinality of Rk.A) after semi-join Rk<B=B]Rp, instead of OC(Rk.A) that is relevant in the reduction of Rj. A_LIST(Rj) is modified so the correct C(Rk.A) that applied in the reduction of Rj is present. C(Rk.A) will now be used for removing the effect of a semi-join which is considered redundant in calculating the correct state of a relation.

We also have to modify the updating method. In general, if there is a non-NULL intersection between A_LIST(Rj) and A_LIST(Ri), we may have different cardinalities associated with the same relations in the two lists. We need to find out how A_LIST(Rj) should be updated if the CSJ is added to G. To this end, Lemmas 2 and 3 are developed below.

Consider two A_PATHs which terminate at Rj. Suppose the two A_PATHs intersect at some other nodes (relations) as well. Then

there must be a node, Rk, such that the two sub-A_PATHs from the node to Rj, do not intersect. In other words, Rk succeeds all other nodes in the intersection. Thus, Rk is defined to be the nearest common predecessor (abbreviated NCP) on the two A_PATHs with respect to Rj.

G and G' in Figures 1 and 2, respectively, are identical except that semi-join Rn<A=A]Rk has been removed from G to make G'. Rk is the NCP with respect to Rj. All semi-joins shown in G and G' are on attribute A. In G', A_PATH1 is an A_PATH from Rn to Rj and A_PATH2 is an A_PATH from Rk to Rj. Other semi-joins (not shown in G or G') on different attributes, may occur on the relations in A_PATH1 or A_PATH2 in G or G'.

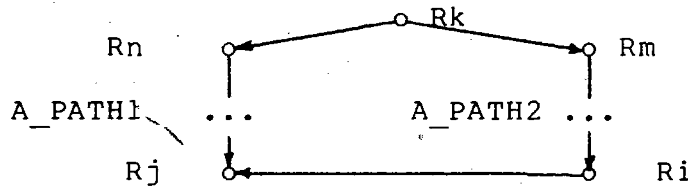



Figure 1: Lemma 2, graph G

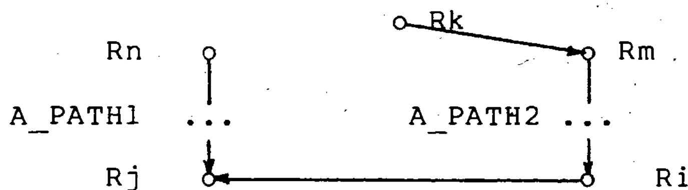



Figure 2: Lemma 2, graph G'

We assume that the cardinality of Rk[A] just before semi-join Rn<A=A]Rk takes place is greater than the cardinality of Rk[A] right before semi-join Rm<A=A]Rk takes place. (i.e. Rn<A=A]Rk occurs first in G and Rm<A=A]Rk occurs sometime later.)

LEMMA 2: Rj[A], after A_PATH1 and A_PATH2 in G have been executed, is identical to Rj[A] after A_PATH1 and A_PATH2 in G' have been executed.

PROOF :

In G, for semi-join Rn<A=A]Rk we have values of A, Rk[A], sent to Rn and Rk'[A] $\subseteq$ Rk[A] sent to Rm on semi-join Rm<A=A]Rk. After Rn<A=A]Rk is executed, we have at Rn, (Rn[A] ∩ Rk[A]).

In G', we can view Rn at the start as comprising the union of two relations: Rn(1), whose column A is (Rn[A] - Rk[A]), and Rn(2), whose column A is (Rn[A] ∩ Rk[A]). Clearly, those values of A at Rn in G after Rn<A=A]Rk will be exactly the same values of A in Rn(2)[A] in G', as defined above.

When A_PATH1 in G' is executed, Rn(1)[A] and Rn(2)[A] are sent from Rn to be joined with those relations along A_PATH1. If Rn(1)'[A] and Rn(2)'[A] are the data transmitted to Rj via A_PATH1 in G', we will have Rn(1)'[A] $\subseteq$ Rn(1)[A] and Rn(2)'[A] $\subseteq$ Rn(2)[A]. Moreover Rn(1)'[A] is disjoint from Rk[A].

After A_PATH1 in G has been executed, if Rn'[A] is transmitted to Rj then clearly Rn'[A] is exactly the same as Rn(2)'[A], since originally (Rn[A] ∩ Rk[A]) in G and Rn(2)[A] in G' were the same and A_PATH1 in G and G' from Rn to Rj have exactly the same semi-joins.

When A_PATH2 in G and G' is executed, Rk'[A] is sent to Rm and eventually a subset of Rk'[A] reaches Rj. Let the values of A sent on semi-join Rj<A=A]Ri be Rk''[A] $\subseteq$ Rk'[A] in both G and G' (since A_PATH2 in G and G' is the same).

Since Rn(1)'[A] at Rj in G' is disjoint from Rk[A] and hence disjoint from Rk''[A] (since Rk''[A] $\subseteq$ Rk'[A] $\subseteq$ Rk[A]) those values of A in Rn(1)'[A] will be eliminated from Rj on semi-join, Rj<A=A]Ri. (For ease of argument we are assuming here that Rj<A=A]Ri is the last semi-join to occur in both G and G'. Our result will be the same if this is not true.) Hence, in G the final state of Rj[A] will be (Rn'[A] $\cap$ Rk''[A]) and the final state of Rj[A] in G' will be (Rn(2)'[A] $\cap$ Rk''[A]). Since Rn'[A] is exactly the same as Rn(2)'[A], the final states of Rj[A] in G and G', are identical.

It is also easy to show that the final state of the whole relation Rj in G in G' will also be identical. Roughly speaking, the lemma shows that G and G' have identical 'reducing effect' on Rj. Comparing G with G', it is obvious that the 'reducing effect' of Rk is delivered via A_PATH2. Thus, we call A_PATH2 the 'effective' A_PATH from Rk to Rj. Lemma 2 also applies when there are more than two A_PATHs from Rk to Rj. The 'effective' A_PATH from Rk to Rj is always the one which includes the latest semi-join starting at Rk.

We can further generalize Lemma 2 so that relation Rk in G does not necessarily have to be an NCP with respect to Rj. This can be obtained by recursively applying Lemma 2. Let Rp be a relation in G which has two A_PATHs leading to Rj. If Rp is an NCP with respect to Rj then Lemma 2 applies. Thus, suppose the two A_PATHs first intersect at a point, say Rj' (Rj' $\neq$ Rj), then Lemma 2 applies to Rp and Rj', as Rp is now an NCP with respect

to Rj', and there is one and only one effective A_PATH from Rp to Rj'.

Consider the CSJ, Rj<A=A]Ri where A_LIST(Rj) and A_LIST(Ri) have a non-NULL intersection. This means there is at least one node say Rk, which has A_PATHs leading to Rj and Ri, respectively. With the addition of the CSJ to G, Rk will have two A_PATHs to Rj. Lemma 2 (and its extensions) applies even when Rk = Ri. However, the same lemma does not apply to the situation when Rk = Rj; in this case there is a cycle which includes Rj. Lemma 3 will deal with this situation.

To be precise, we have G and G' in Figures 3 and 4, respectively, which are identical except that semi-join Rm<A=A]Rj has been removed from G to make G'. All semi-joins shown in G and G' are on attribute A. A_PATH1 in G is an A_PATH from Rj to Rm and back to Rj again. A_PATH1 in G' is an A_PATH from Rm to Rj. If any other semi-joins on different attributes occur to the relations in G, they also occur to the relations in G'. These semi-joins are not shown in Figures 3 and 4.

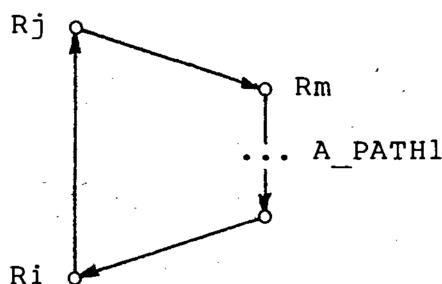



Figure 3: Lemma 3, graph G

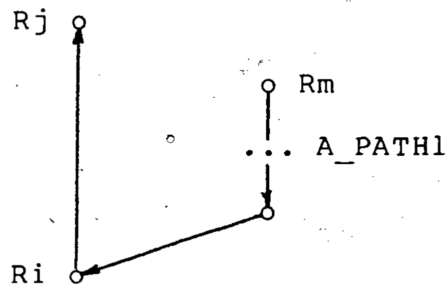



Figure 4: Lemma 3, graph G'

LEMMA 3: Rj[A] after A_PATH1 in G has been executed is identical to Rj[A] after A_PATH1 in G' has been executed.

The proof of Lemma 3 is omitted since the argument is very similar to the proof for Lemma 2.

The significance of Lemma 3 is that effective A_PATHs have no cycles. Combining Lemmas 2 and 3, we have:

THEOREM 1: There is at most one effective A_PATH (involving no cycles), between any pair of relations in G.

With this theorem, we can define a structure, A_TREE, such that only the effective A_PATHs on attribute A to Rj are in A_TREE(Rj). Rj is the root. A_TREE(Rj) contains no cycles and there is only one occurrence of each relation in A_TREE(Rj).

## 4.3 THE EFFECT OF SEMI-JOINS ON NON-JOINING ATTRIBUTE CARDINALITIES

There is no agreement in the literature regarding how cardinalities of attributes other than A are updated with

respect to the semi-join, say R<A=A]S, reducing R.A. [Hevn79a] maintains that they should remain unchanged, while [Yu79] argues that they should be reduced in the same manner as R.A by applying the selectivity of the semi-join to the current cardinality of each attribute. [Good79] uses an approximation of Yao's function [Yao77] which depends on the selectivity of the semi-join. We feel that [Good79] has the most realistic approach but we have determined that there are several unrealistic assumptions utilized in his approach. [Good79]'s approach is explained next along with our reservations about his method.

Algorithm AP [Good79] (implemented on SDD-1, an experimental distributed database system by the Computer Corporation of America) analyzes the effect on other attributes in R as a 'hit ratio' problem, given Assumption 1. If we are given n = cardinality of R "objects", distributed uniformly over m = cardinality of R[B] "colours" (where $B \neq A$ ) then the hit ratio problem may be stated as, "How many colours are we expected to hit if we randomly select r of the objects?".

The answer is given by [Yao77]:

$$Y(m,n,r) = m * (1 - \prod_{i=1}^{r} [(nd - i + 1)/(n - i + 1)]$$

where d = 1 - 1/m.

The computation of this function is time consuming, so in practice [Good79] approximates Y(m,n,r) by:

$$AY(m,n,r) = \begin{cases} r & , \text{ for } r < m/2 \\ (r+m)/3 & , \text{ for } m/2 \leq r < 2m \\ m & , \text{ for } 2m \leq r \end{cases}$$

There are two problems with Y(m,n,r) and the way in which [Good79] utilizes it, as Luk [Luk80] has shown. It is quite clear that the function Y(m,n,r) is not linear in either m, n or r. While the values of n and m are known here, r is a random variable of some unknown distribution. [Luk81] has shown that in the literature the average value of r is invariably treated as a fixed value in Y(m,n,r), (as [Good79] does), instead of taking the average of all function values over the distribution of r. Given m, when the upper bound of the range of r is between 2 and 10 times as great as m, the error introduced in calculating Y(m,n,r), when using a fixed value for the average value of r, is greater than 10%.

[Good79] implicitly assumes that if we have m distinct values in any column (attribute) then there are exactly n/m occurrences of each distinct value in R in A. Luk has pointed out that this is an extremely unlikely situation and is only likely to occur when R is the cartesian product of the set of all attribute values for each attribute in R. Luk assumes the number of tuples in R with the same value of attribute B is a random variable and is distributed according to Zipf's distribution [Zipf49]. Luk has shown that, using [Good79]'s assumption, errors in estimating Y(m,n,r) are often greater than 50% and sometimes as large as 100%.

Clearly, if one repeatedly applies results from Y(m,n,r) without correcting the two problems mentioned above, the estimation of new database states through the use of this function will be erroneous.

We have incorporated Luk's result in this work. Luk's result is also computationally time consuming, so in practice it may be approximated by:

$$AL(m,n,r) = \begin{cases} r/2 & , \text{ for } r \le m/2 \\ .3r+.1m & , \text{ for } m/2 < r \le m \\ (r+m)/5 & , \text{ for } m < r \le 2m \\ .1r+.4m & , \text{ for } 2m < r \le 4m \\ .04r+.6m & , \text{ for } 4m < r \le 9m \\ m & , \text{ for } 9m < r \end{cases}$$

AL(m,n,r) is then the estimating function to determine the new cardinalities of attributes, other than the joining attribute, in a relation being reduced by a semi-join.

## 4.4 BENEFIT AND UPDATE CALCULATION ALGORITHMS

We now make use of the results from the last section to correctly calculate the cost and benefit of a semi-join Rj<A=A]Ri. To do this we must first identify effective A_PATHs from other relations in G to Rj. The A_TREE data structure will be used for this purpose.

A straight-forward method to estimate the state of a relation Rj is to work with A_TREE(Rj), B_TREE(Rj),... etc, for all the join attributes, A, B,... etc. The effects of the semi-joins in all these lists are calculated according to the sequence number of the semi-joins.

This process can be considerably simplified if we make an assumption on the commutativity of the estimating functions. Let f be the function to estimate the cardinality of Rj[A] after some semi-join, say Rj<B=B]Ri (SJ-1), with selectivity Si. Let g be the function to estimate the cardinality of Rj[A] after some

semi-join, say Rj<A=A]Rk (SJ-2), with selectivity Sk. If the current cardinality of Rj[A] is denoted by CC(Rj.A), then the new CC(Rj.A) after SJ-1 will be f(CC(Rj.A),Si) and independently the new CC(Rj.A) after SJ-2 will be g(CC(Rj.A),Sk). We now assume that the estimated cardinality of Rj[A] after SJ-1 and then SJ-2 is the same as after SJ-2 and then SJ-1, or equivalently,

g(f(CC(Rj.A),Si),Sk) =
f(g(CC(Rj.A),Sk),Si)

Now Sk = CC(Rk.A)/DC.A and according to (NEC),

g(CC(Rj.A),Sk) = CC(Rj.A) *
CC(Rk.A)/DC.A

so that we have:

f(CC(Rj.A)*CC(Rk.A)/DC.A, Si) =
f(CC(Rj.A),Si)*CC(Rk.A)/DC.A

Consider now an effective A_PATH from R1 to R3. For convenience, we assume there is only one semi-join applied to R2 and R3 with a join attribute other than A and selectivities S2 and S3.

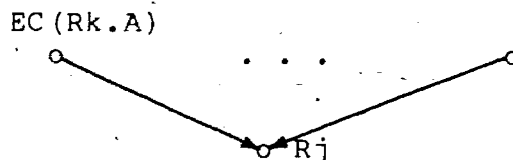
Since the order of execution of the semi-joins is unimportant, CC(R2.A) is equal to

f(OC(R2.A),S2) * OC(R1.A)/DC.A

and the CC(R3.A) is equal to

f(OC(R3.A),S3) *
f(OC(R2.A),S2) *
OC(R1.A)/DC.A

To generalize, if we have n semi-joins on a branch of A_TREE(Rj), this branch is equivalent to the following tree:




where EC(Rk.A) (abbreviation for effective cardinality), $1 \leq k \leq n$, represents the cardinality of Rk after all the semi-joins with join attributes other than A have been applied to Rk. If there is no such semi-join on Rk, EC(Rk.A) is the OC(Rk.A). Note also that $1 \leq j \leq n$.

By Theorem 1, all branches of A_TREE(Rj) are independent, so that an A_TREE(Rj) with n semi-joins is equivalent to another A_TREE(Rj) with n parallel semi-joins. The advantage of this representation is that the effect of each semi-join on Rj is 'normalized' so that the removal of the effect of a certain semi-join is now a trivial task. Note the similarity of this representation and the one for the single join attribute case.

We now modify the A_LIST(Rj) to reflect this representation of A_TREE(Rj):

Rj : (EC(R1.A),I1),...,(EC(Rn.A),In)

where Ii, $1 \leq i \leq n$, represents the semi-join number assigned for that semi-join (in RHO). If i=j, then Ij = Infinity (abbreviated INF). Thus,

$$CC(Rj.A) = EC(Rj.A) * \prod_{\substack{i=1 \\ i \neq j}}^{n} EC(Ri.A)/DC.A$$

This formula is similar to the one derived for the single attribute case discussed earlier in this chapter. The algorithm to update the A_LISTs is given below. Initially A_LIST(Rj) will

contain one element, i.e. Rj(EC(Rj.A),INF) where EC(Rj.A) = OC(Rj.A).

```
ALGORITHM UPDATE:
Let the semi-join being added to RHO be: p,Rj<A=A]Ri
  NON-JOINING ATTRIBUTE UPDATE:
  IF (B is an attribute in Rj)
     THEN  B_LIST(Rj) will contain Rj(EC(Rj.B),INF).
           Update EC(Rj.B) here to:
             f(EC(Rj.B),S), S being the
             selectivity of the semi-join.
  JOINING ATTRIBUTE UPDATE:
  IF (A_LIST(Rj) ∩ A_LIST(Ri) = NULL)
     THEN  A_LIST(Rj) := A_LIST(Rj) U  A_LIST(Ri)
     ELSE  A_LIST(Rj) := A_LIST(Rj) U  A_LIST(Ri)
       SUCH THAT:
         FOR EACH RELATION, SAY:
           Rk(EC(Rk.A),v) (an element of
           A_LIST(Rj)) and Rk(EC(Rk.A),u)
           (an element of A_LIST(Ri))
           in the intersection, then:
             IF (u <= v)
             (i.e. u is an earlier SJ)
                THEN
                  Leave Rk(EC(Rk.A),v) in A_LIST(Rj).
                ELSE
                  Replace Rk(EC(Rk.A),v) in A_LIST(Rj)
                  with Rk(EC(Rk.A),u) from A_LIST(Ri).
```

At any step from the beginning, we know the current cardinalities of Ri.A (i.e. CC(Ri.A)), Rj.A (i.e. CC(Rj.A)) and Rj (i.e. CC(Rj)). If we use the above algorithm to update the database state after the CSJ Rj<A=A]Ri has been added to RHO, we have a new cardinality of Rj.A, say NC(Rj.A). The effective selectivity of the semi-join is then NC(Rj.A)/CC(Rj.A). Thus

```
COST (of the CSJ) = CC(Ri.A), and
BENEFIT (of the CSJ) =
     (CC(Rj) * a) - ((CC(Rj)*NC(Rj.A)/CC(Rj.A)) * a)
```

where 'a' is the number of attributes in Rj.

## 4.5 IMPLICATIONS FOR A DISTRIBUTED DATABASE SYSTEM MODEL

The main contribution of this chapter is the definition of a realistic distributed database system model (abbreviated DDSM). Since we are mainly concerned with query processing, our work has concentrated on the accurate estimation of new database states when semi-join operations are executed. We have developed an estimating function to correctly determine, after a semi-join is executed, the new cardinality of both the relation being reduced and the joining attribute in that relation. We have developed an estimating function to determine the new cardinalities of the non-joining attributes in the relation being reduced. Algorithm UPDATE utilizes these estimating functions to update the database state. Clearly the information and assumptions these estimating functions utilize define a particular model of a distributed database system. The ultimate goal is to have semi-joins perform in practice as they do in one's model of the distributed database system. It is felt that our model is a realistic one.

There are two ways that the estimating functions will be used in this work.

In algorithms that generate semi-join programs, the estimating functions may be utilized to determine the potential benefit for each semi-join considered and once a semi-join is chosen, to update the database state via algorithm UPDATE. In this way we can develop and test the performance of algorithms that use semi-join tactics under our DDSM.

If we already have a semi-join program and wish to

determine how it performs under our DDSM then we can re-execute this semi-join program. The way to accomplish this is to consider each semi-join in the semi-join program in order. Let the ith semi-join be R<A=A]S. Then the cost of R<A=A]S can be determined by examining the current database state for the cardinality of S.A. We execute the semi-join by updating the database state; using algorithm UPDATE to determine the new cardinalities of R, R.A and the other non-joining attributes in R. If we then total the costs determined for each of the semi-joins in the semi-join program, we have 're-cost' this semi-join program. We can thus check the performance of semi-join programs produced by other published algorithms.

# 5. PREPROCESSING STRATEGIES FOR DISTRIBUTED QUERIES

## 5.1 INTRODUCTION

In the general distributed query environment, the semi-join operation has been used as a preprocessing strategy for queries. The objective of semi-join preprocessing is to reduce the total amount of data required to be transmitted on the network by first reducing the cardinalities of distributed relations using semi-joins and then transmitting the resultant relations to the result node. Algorithms that utilize semi-join tactics produce as output a semi-join program that is to be executed on the network.

In this chapter we first present two existing query preprocessing algorithms, algorithm AP [Good79] and algorithm HEVNER [Hevn79a]. Algorithm AP is currently implemented on SDD-1 an experimental distributed database system. Algorithm HEVNER is a query processing algorithm proposed in the Ph.D thesis of A. Hevner [Hevn79a]. We next present algorithm BLACK, a query preprocessing algorithm that represents a result from our work in this area. We then present a 're-organization' algorithm called WS [Luk80] that, when given a semi-join program as input, produces a new semi-join program that is guaranteed to have a new cost less than or equal to the cost of the original semi-join program. Algorithm WS is used in conjunction with algorithm BLACK to produce lower cost semi-join preprocessing strategies. The last section of this chapter gives an example query along with the semi-join programs produced by the various

algorithms on this query.

It should be noted again that all these algorithms are heuristic in nature since the determination of an optimal semi-join program is an NP-HARD problem. To date, no algorithms producing optimal semi-join programs are known.

## 5.2 THE INITIAL FEASIBLE SOLUTION

The 'initial feasible solution' (abbreviated IFS) for a given query is to first do all initial local processing then transmit all relations to the result node.

## 5.3 ALGORITHM HEVNER

Algorithm HEVNER was called Algorithm General: Total Time in [Hevn79a]. Hevner and Yao [Hevn79b] were the first researchers to determine an optimal solution for solving simple queries (queries with only one joining attribute and no output attributes other than the joining one). Hevner proved that the optimal solution for more general queries is an NP-HARD problem. Hevner, in algorithm HEVNER, utilized tactics developed in the solution for simple queries. For these queries, the relations are ordered according to their cardinalities. Starting from the smallest one, a relation is sent serially to the next smallest one to perform the semi-join. Thus, only the transmission cost at every step is guaranteed to be minimal, a minimal size relation is also created, which in turn guarantees minimal transmission cost at the next step. Hevner utilizes the concept

of a relation schedule , i.e. a sequence of semi-joins to be executed in linear order to that relation, as a preprocessing step for that relation. The construction of a relation schedule is based on simple query tactics. This is the main reason why a schedule results in minimum transmission cost. A problem with this tactic, though, is that relation schedules are independent. That is, if R2 is reduced in cardinality in R1's schedule, in the schedule for R2 no notice of this fact is made. Algorithms AP and BLACK always make note of, and utilize, reductions of relations. Also even though HEVNER uses simple query tactics, it is so constructed that optimal solutions for simple queries do not occur and in fact AP and BLACK perform better on simple queries than does HEVNER.

Before introducing algorithm HEVNER, the data structures and terminology used in the algorithm will be explained.

For each attribute A in the query we can define its 'simple query solution' in the following way. For those relations, say {R1,...,Rm}, with attribute A, create a schedule S such that the relations are ordered in increasing order of the cardinality of attribute A. (We are assuming here that all relations are at different nodes in the network.)

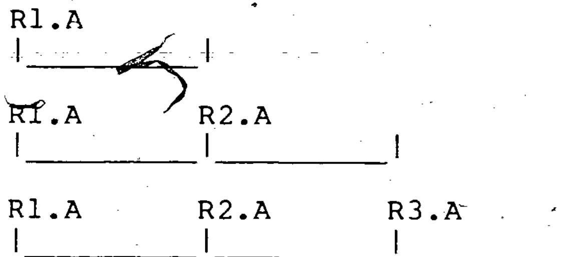
For instance, if this ordering happened to be R1,R2,...,Rm then the simple query schedule (solution) is:

```
R1.A        R2.A                       Rm.A
|__________|__________|     . . .     |__________|
```

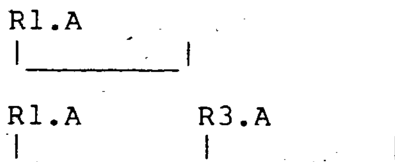
The schedule is to be executed from left to right. The semi-joins are R2<A=A]R1, R3<A=A]R2, ... Rm<A=A]Rm-1. The last transmission is to the result node. R1.A has the smallest

cardinality of R1.A,R2.A,...,Rm.A and the above solution is the optimal solution if this is a simple query. Attribute A in R2 gets reduced in cardinality by the semi-join, R2<A=A]R1. The reduced R2.A reduces R3, and so on.

For each attribute, CASE 1 serial schedules are formed. For example, let there be three relations, R1, R2 and R3, each at different nodes in the network, and let the cardinality order be R1.A < R2.A < R3.A. Then the CASE 1 serial schedules are defined as follows:




If we are trying to determine the relation schedule to relation Ri in HEVNER, we first form the CASE 2 serial schedules for relation Ri. For example the CASE 2 serial schedules for relation R2 from the above example will be:




This is done by eliminating R2.A's transmission from the CASE 1 serial schedules and eliminating duplicate schedules. Algorithm HEVNER then utilizes these CASE 2 serial schedules in forming relation schedules.

We now present algorithm HEVNER.

ALGORITHM HEVNER

1. Generate candidate relation schedules. Isolate each of the joining attributes and consider each to define a simple query with an undefined result node. Form CASE 1 serial schedules.
2. Select the best candidate schedule. For each relation form the CASE 2 serial schedules for each attribute. Each schedule in the set of CASE 2 serial schedules is considered to be a 'candidate schedule' to the relation for that attribute. If the cost of a candidate schedule plus the final transmission to the result node has less cost than the initial feasible solution for that relation and is the least cost candidate schedule then save that candidate schedule for that attribute for that relation.
3. Integrate the schedules. If only one schedule has been saved for relation, Ri, then this is the schedule to relation Ri. Otherwise the saved schedules need integration. This is done by Procedure TOTAL given below.
4. Remove schedule redundancies. Eliminate schedules for relations which have been transmitted in the schedule for another relation.

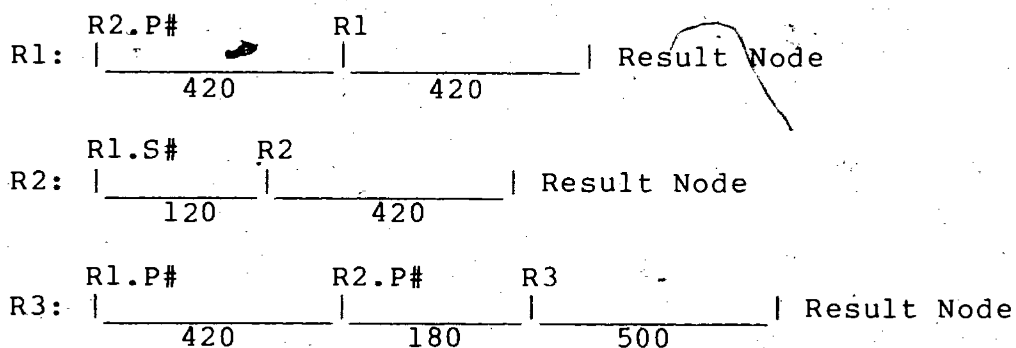
Procedure TOTAL

1. Candidate schedule ordering. For each relation, say Ri, order the saved schedules in increasing order of total cost

(i.e. the cost of the schedule plus the cost of the transmission of Ri to the result node).

2. If S1,S2,...,Sn are the saved schedules to relation Ri, in order of increasing total cost, then form the integrated schedules SI1,SI2,...,SIn, which consist of the parallel transmission of the saved schedules to relation Ri, such that S1 is the only schedule in SI1, S1 and S2 are the only schedules in SI2, and Si (i≤j) are the only schedules in SIj. Select the integrated schedule SIj that results in the minimal total time value.

Since HEVNER produces parallel independent relation schedules, we have converted HEVNER's output into a semi-join program using the following process. We will present the algorithm using an example. Consider the following relation schedules:




The length of a gap signifying a transmission (or semi-join) is meant to be proportional to the transmission cost labelled on that gap. The final transmission in each relation schedule is the transmission of that relation to the result node. This

transmission is ignored in forming the semi-join program. We generate a semi-join program from these relation schedules by first examining all schedules and choosing the semi-join with the least distance from the beginning of a relation schedule. This process effectively times the occurrence of the semi-joins with respect to the relation schedules. This semi-join is added to RHO, the semi-join program. In this case, the semi-join first chosen is R2<S#=S#]R1. This semi-join is noted as being used. The next semi-join chosen is the one with the next least distance from the beginning of the relation schedules. This semi-join is R1<P#=P#]R2. Then R2<P#=P#]R1 and R3<P#=P#]R2 are chosen in order. The semi-join program, RHO, is then

1, R2<S#=S#]R1
2, R1<P#=P#]R2
3, R2<P#=P#]R1
4, R3<P#=P#]R2

If any semi-joins were duplicated in RHO the later duplicates in RHO would be eliminated. Duplicate semi-joins tend to increase the cost of RHO.

By adding a timing consideration for the execution of semi-joins in Hevner's relation schedules, intermediate reductions of relations are now being utilized hence there will be a decrease in the cost of his query processing strategies. The basic principle of using relation schedules in his algorithm has been retained and effectively we have introduced an enhancement to his algorithm.

## 5.4 ALGORITHM AP

Algorithm AP (Access Planner) [Good79] is a greedy optimization algorithm since the semi-joins it chooses to include in RHO are always the least cost semi-joins as determined by the SDD-1 model of a distributed database system. There are only two requirements for a semi-join to be included in RHO. The first is that it must have the least cost in terms of the amount of data transmitted and the second is that the cost must be less than or equal to the benefit. The benefit is defined as the amount of data reduction at the relation being reduced by the semi-join. Clearly with these conditions the semi-join program produced will always lead to a preprocessing strategy with total cost (the cost of executing the semi-joins plus the final transmissions to the result node) less than or equal to the initial feasible solution.

The semi-joins AP considers for inclusion in RHO are those implied by the transitive closure of the join clauses expressed in the initial query, as discussed earlier. A semi-join once included in RHO is not considered again for possible addition to RHO.

Before algorithm AP is introduced, the data structures used in the algorithm are discussed.

RHO is the resultant semi-join program to be executed on the distributed database system. OMEGA is the set of semi-joins that AP considers. When a semi-join is included in RHO it is eliminated from OMEGA and OMEGAprofitable. OMEGAprofitable is the set of semi-joins from OMEGA which have their cost less than

or equal to their benefit. The algorithm utilizes the current database state for determining the current cost and benefit of semi-joins. This consists of the current relation cardinalities and the cardinalities of all attributes in those relations. When a semi-join is chosen to be included in RHO the current database state is updated to reflect the effect of that semi-join on the database state. Clearly only the state of one relation needs to be updated for a given semi-join. When a semi-join is included in RHO, OMEGAprofitable must be updated on those semi-joins which reduce the relation just reduced and those semi-joins emanating from the relation just reduced to determine if those semi-joins should still be in OMEGAprofitable and to update their cost and benefits.

ALGORITHM AP

```
STEP 1 - INITIALIZATION
  a. RHO := null program
  b. OMEGA := the set of legal semi-joins
  c. OMEGAprofitable := those semi-joins from OMEGA
                          such that cost <= benefit
STEP 2 - MAIN LOOP
  WHILE OMEGAprofitable is not null DO
    Append to RHO the semi-join in OMEGAprofitable
      that has the least cost over all semi-joins
      in OMEGAprofitable.  Remove this semi-join
      from OMEGA and OMEGAprofitable.
    Update the database state.  Update OMEGAprofitable.
STEP 3 - TERMINATION
  The reduced relations are now transmitted to the
    result node.  We assume here that the result
    node is none of the relations referenced in a
    join clause in the query and all reduced
    relations are transmitted to this node.
```

The main problem with this algorithm is its method for updating the database state. Under the distributed database system model used in SDD-1 the history of previous semi-joins in RHO is ignored when doing database state updates. As shown

earlier this leads to erroneous updates and leaves the quality of the semi-join programs produced in question.

Recently (Dec. 1981) a new version of algorithm AP (called OPT) has been published [Good81]. OPT incorporates the solution for the semi-join history problem. The primary heuristic for determining the next semi-join to be included in RHO has also been changed. Instead of choosing the least cost semi-join in OMEGAprofitable, OPT chooses the semi-join with the greatest profit (i.e. the benefit minus the cost of the semi-join). In Chapter 6 we present results showing the performance of both algorithm's AP and OPT.

## 5.5 ALGORITHM BLACK

Algorithm BLACK is similar in style to algorithm AP, however there are three important differences. The history of previous semi-joins that have been included in RHO is not ignored when calculating the benefit of a semi-join. The cardinality of non-joining attributes in a relation being reduced by a semi-join are updated according to AL(m,n,r) (introduced in Chapter 4). We use the simple query strategy as a heuristic for more general queries. We can identify two heuristics. The first is transmit the least amount of data at each step (i.e. choose the least cost semi-join). The second heuristic is to choose a semi-join that, when it is executed, produces the smallest cardinality attribute. These two heuristics are interdependent; that is, the second heuristic provides the basis for the first. We utilize these heuristics in

algorithm BLACK, and in this sense we say that algorithm BLACK takes a middle position between algorithms AP and HEVNER. These heuristics are discussed in more detail below.

When choosing the least cost semi-join from OMEGAprofitable, there will likely be several semi-joins with the same least cost. When we compute the transitive closure of the join clauses in a query, it is likely that a relation will be involved in joins with two or more relations on the same attribute; hence adding semi-joins of equal cost to OMEGAprofitable. Algorithm BLACK chooses the semi-join from this set of least cost semi-joins that produces the smallest cardinality attribute. (Note that we have made the assumption in Chapter 2 that the attribute size and cardinality are equivalent and hence the cost of a semi-join, say R<A=A]S, is the same as the cardinality of S.A.) Further, if R<A=A]S is a least cost semi-join in OMEGAprofitable, then the cardinality of R.A is guaranteed to be less than or equal to the cardinality of S.A after the semi-join is executed. This is true since the semi-join R<A=A]S is known to be cost beneficial. The equality holds only when C(R.A)=DC.A or when R.A before the semi-join is executed is a subset of S.A. If we assume neither of these conditions is true then we are guaranteed that the next semi-join chosen by AP or BLACK will be a semi-join emanating from relation R. This is true since the only effect on the database state when R<A=A]S is executed is the reduction of relation R, hence the cost of all semi-joins in OMEGA and OMEGAprofitable remain the same except for those from relation

R. We can recursively apply this argument for the relation being reduced as long as the semi-joins in RHO and the CSJ have no common history and there are still semi-joins in OMEGAprofitable from the last relation that has been reduced.

This process occurs in algorithms AP and BLACK; however, in BLACK we have added a new heuristic. We do not arbitrarily choose a semi-join to be included in RHO from the set of least cost semi-joins as AP does. We choose the semi-join to a relation that will produce the smallest cardinality attribute in that relation (joining or non-joining attribute) over the set of least cost semi-joins. The rationale for this choice is based on the previous discussion. The next semi-join chosen will likely be from the relation with the smallest cardinality attribute because semi-joins on this attribute from this relation will be the least cost semi-joins in OMEGAprofitable (if there are any). The next semi-join chosen will then have less data communication cost and greater benefit at the relation being reduced.

Algorithm UPDATE is utilized to take into account the history of previous semi-joins in RHO. When a semi-join is added to RHO, UPDATE may be utilized to determine the new database state. When the benefit of a semi-join is to be determined either when updating OMEGAprofitable or when examining semi-joins in OMEGA, UPDATE may be utilized to determine the new hypothetical database state if this semi-join were executed and hence its benefit.

Algorithm BLACK is now presented.

ALGORITHM BLACK

STEP 1 - INITIALIZATION

```
  a. RHO := null program
  b. OMEGA := the set of legal semi-joins
  c. OMEGAprofitable := those semi-joins from OMEGA
                        such that cost <= benefit
```

STEP 2 - MAIN LOOP

```
  WHILE OMEGAprofitable is not null DO
    Append to RHO the semi-join in OMEGAprofitable
      that has the least cost over all semi-joins
      in OMEGAprofitable.  (If there is more than
      one semi-join in OMEGAprofitable with the
      same least cost then the semi-join appended
      to RHO is the one which produces the
      smallest cardinality attribute in the relation
      being reduced.)
      Remove this semi-join from OMEGAprofitable
      and OMEGA.
    Update the database state using algorithm
      UPDATE (from Chap. 4).  Update OMEGAprofitable
      using algorithm UPDATE.
```

STEP 3 - TERMINATION

```
  The reduced relations are now transmitted to the
    result node.
```

## 5.6 ALGORITHM WS

Algorithm WS [Luk80] is an algorithm that seeks to improve a semi-join program produced by some arbitrary heuristic; the program is transformed into one with non-increasing cost and non-decreasing benefit. Let RHO be the input semi-join program and RHO', the output semi-join program from algorithm WS. Then RHO' satisfies the following conditions:

1. RHO' and RHO give the same answer to any given query.
2. RHO' has a cost no greater than RHO for all instances.
3. RHO' has a benefit no less than RHO for all instances.
4. RHO' is produced given only the program RHO; no additional information, such as cardinalities of the attributes and relations, is available to algorithm WS.

Furthermore, it has been proved that the improvement of RHO' over RHO is optimal in the sense that there will not be another RHO'' which is better than RHO' and still satisfies conditions (1) to (4). For a more detailed description of algorithm WS see [Luk80].

## 5.7 ORDER OF COMPLEXITY OF THE ALGORITHMS

Algorithms HEVNER, AP and BLACK all run in the worst case complexity of O(NA*(NR**2)) where NA is the number of attributes and NR is the number of relations in the query. This time complexity reflects the number of possible semi-joins when we have NA attributes and NR relations in a query. The time complexity of algorithm WS is O(NR**3).

## 5.8 AN EXAMPLE

An example is now presented to illustrate the results produced by algorithms HEVNER, AP, BLACK and WS.

The example query has the following qualification.

R1[P#=P#]R2 AND R1[P#=P#]R3 AND
R1[S#=S#]R2 AND R2[A#=A#]R3

The database state is initially:

| | RELATION SIZE | JOINING ATTRIBUTE CARDINALITIES | | |
|---|---|---|---|---|
| | | P# | S# | A# |
| R1: | 1000 | 400 | 100 | |
| R2: | 2000 | 400 | 450 | 100 |
| R3: | 3000 | 900 | | 300 |

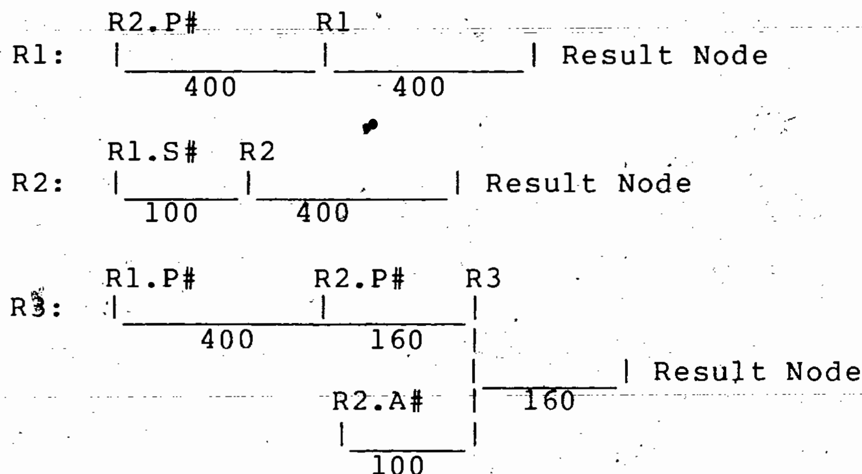
where the domain cardinalities for attributes P#, S# and A# are, respectively, 1000, 500 and 300. R1, R2 and R3 are at different nodes in the network. The cost in data communication for the

initial feasible solution (IFS) for this query would be, 1000+2000+3000=6000. The results presented below may be compared to the IFS cost to see how well the semi-join preprocessing strategies perform.

The results produced from this query are given in the next sections.

5.8.1 HEVNER'S RESULT

Algorithm HEVNER when given this query produces the following relation schedules.




The total cost of these schedules (ignoring Hevner's data transmission cost constant) is 2120. The semi-join program we generated from these relation schedules is:

1, R2<S#=S#]R1
2, R3<A#=A#]R2
3, R1<P#=P#]R2
4, R2<P#=P#]R1
5, R3<P#=P#]R3

This semi-join program, re-cost under our DDSM, has a total cost of 836 including transmissions to the result node.

## 5.8.2 AP'S RESULT

Algorithm AP produces the following semi-join program for the example query.

1, R3<A#=A#]R2
2, R2<A#=A#]R3
3, R2<S#=S#]R1
4, R1<S#=S#]R2
5, R2<P#=P#]R1
6, R1<P#=P#]R2
7, R3<P#=P#]R1
8, R2<P#=P#]R3

This semi-join program with a total cost of 485 under algorithm AP's DDSM when re-cost under our DDSM has a total cost of 821. The large difference in these costs is mainly due to algorithm AP ignoring the history of previous semi-joins in RHO.

## 5.8.3 BLACK'S RESULT

Algorithm BLACK produces the following semi-join program for the example query.

1, R2<S#=S#]R1
2, R3<A#=A#]R2
3, R1<S#=S#]R2
4, R1<P#=P#]R2
5, R3<P#=P#]R1
6, R2<P#=P#]R3
7, R1<P#=P#]R3
8, R2<A#=A#]R3
9, R3<P#=P#]R2

The total cost of this semi-join program is 571.

When algorithm WS is applied to the above semi-join program, it produces the following semi-join program.

1, R2<S#=S#]R1
2, R3<A#=A#]R2
3, R1<S#=S#]R2
4, R1<P#=P#]R2
5, R3<P#=P#]R1
6, R2<P#=P#]R3
7, R2<A#=A#]R3
8, R1<P#=P#]R2
9, R3<P#=P#]R1

The total cost of this semi-join program is 525.

Algorithm WS has made changes to the last three semi-joins in the semi-join program produced by BLACK. Semi-join 8 from BLACK has been repositioned by WS to become semi-join 7. Semi-join 7, R1<P#=P#]R3, from BLACK has been changed by WS to semi-join 8, R1<P#=P#]R2. Semi-join 9, R3<P#=P#]R2, from BLACK has been changed by WS to semi-join 9, R3<P#=P#]R1. These last two changes by WS are responsible for the cost reduction between the semi-join programs produced by algorithms BLACK and WS.

In summary, we see that algorithm HEVNER and algorithm AP both produce more costly semi-join programs than algorithm BLACK. This greater cost is due to an over-simplified DDSM in HEVNER's case and an incorrect DDSM in AP's case.

# 6. SIMULATION MODEL AND RESULTS

## 6.1 INTRODUCTION

We have developed a simulation program to test the performance of the algorithms introduced in the previous section. This program accepts as input a stream of queries along with a database description for each query. Each algorithm interprets each query and generates a semi-join program. The semi-join program is re-cost under our DDSM and the cost in data communication of the semi-join program is recorded. The program outputs for each algorithm the average total communication cost per query. This value includes the cost of executing the semi-join program and the final transmission of relations to the result node. The cost in data communication on the network of the semi-join program is the sum of the costs for each semi-join in the semi-join program. The cost of a semi-join, say Ri<A=A]Rj, is C(Rj.A). The cost of the final transmission of relations to the result node is the sum of the relation sizes after the semi-join program has been executed on the database.

In the next section we discuss a method of generating random queries and what criteria we place on these queries.

## 6.2 GENERATING RANDOM QUERIES

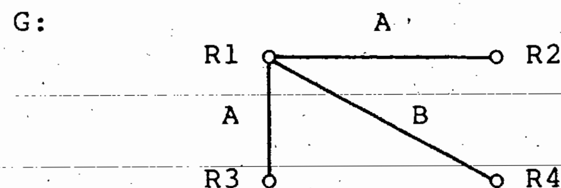
We are interested in how each algorithm performs in a general query environment. To be able to compare the performance of each algorithm, it was necessary to average total communication costs for each query for each algorithm over a

large number of queries. To give some feeling of the range that possible results could have, the number of join clauses in the queries was varied. This variation occurred over three intervals; queries with a minimum number of join clauses, queries with a maximum number of join clauses, and queries with an intermediate number of join clauses. How these queries were generated is discussed next.

6.2.1 VARYING THE NUMBER OF JOIN CLAUSES

Given the number of attributes and relations in a query, it is possible to generate random queries with a minimum, maximum and intermediate number of join clauses. Since there is one relation per node in the distributed database, a graph with the relations as vertices may be used to model a query. The edges in the graph are labelled with attributes. If an edge exists between say R1 and R2 labelled with attribute A then the join clause, R1[A=A]R2, is present in the query. The graph is made a connected graph otherwise the query degenerates into two or more independent queries.

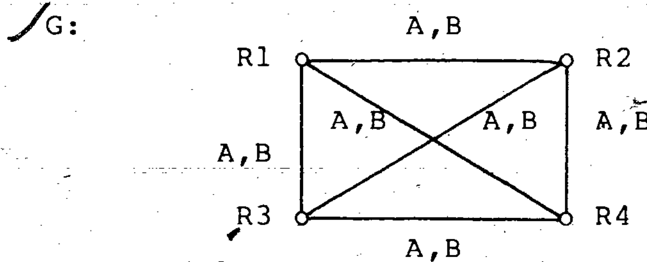
If, for example, we have a query with four relations and two attributes, then a query with a minimum number of join clauses can be expressed as a graph G, as follows:




The join clauses are R1[A=A]R2, R1[A=A]R3 and R1[B=B]R4. The join clause R2[A=A]R3 is also present and is implied by

transitivity. For this number of relations and attributes and graph G, the number of join clauses is minimal. We cannot remove a relation, an edge or an edge label from G without changing the query, disconnecting G and unlabelling an edge, respectively.

We can construct a query with a maximum number of join clauses by allowing G to be the complete graph on four vertices and labelling every edge with every attribute, as follows:




It is easy to see that every join clause possible is represented in G.

The method used for generating these types of queries is described in more detail below. Queries with an intermediate number of join clauses are discussed in more detail below.

### 6.2.1.1 QUERIES WITH A MINIMUM NUMBER OF JOIN CLAUSES

Queries with a minimum number of join clauses are generated by randomly generating minimally connected graphs with the relations as vertices. Attributes are randomly assigned to the edges. If there are more attributes than edges, the extra attributes are randomly assigned to the edges. If there are fewer attributes than edges, the extra edges are randomly assigned attributes.

#### 6.2.1.2 QUERIES WITH A MAXIMUM NUMBER OF JOIN CLAUSES

Queries with a maximum number of join clauses are generated by forming the complete graph with the relations as vertices. Every attribute is assigned to every edge.

#### 6.2.1.3 QUERIES WITH AN INTERMEDIATE NUMBER OF JOIN CLAUSES

Queries with an intermediate number of join clauses are generated by first forming a minimally connected graph with the relations as the vertices. A random number of edges are then added to this graph, ignoring collisions; no more than half the number of edges in the complete graph for this number of vertices are added. The median value of the number of edges added is half the maximum value. Attributes are then assigned to the edges as in queries with a minimal number of join clauses.

### 6.2.2 VARYING THE NUMBER OF RELATIONS, ATTRIBUTES AND AVERAGE SELECTIVITY

For each of the three types of queries above, the number of attributes is varied over the range 1-5, the number of relations is varied over the range 2-9, and the average selectivity of attributes in relations is varied over the values 0.05, 0.10, 0.20, 0.30, 0.40 and 0.50. If AS is the average selectivity then the range of selectivity values randomly generated to arrive at the above means are $0.0 < AS \leq 2AS$.

In the next section we discuss how the database cardinalities for both relations and attributes were randomly generated.

## 6.3 RANDOM DATABASE DESCRIPTIONS

Once the number of relations in a query is known the cardinality of the relations can be randomly generated. The maximum cardinality is set at 100,000. Results from this simulation are quoted relative to each other so absolute values become unimportant.

The domain cardinalities of the attributes are then randomly generated. The cardinalities of the attributes in the relations are randomly generated to a maximum of the domain cardinality for that attribute or the relation cardinality, whichever is smaller.

All random values are generated uniformly.

## 6.4 STEADY STATE CONSIDERATIONS

The simulation program is allowed to test enough queries of a given type so that the cumulative average of the total transmission cost of a query quietens (i.e. reaches steady state where there is little fluctuation in this value).

## 6.5 RESULTS

The major statistic compilied in the simulation is the average cost in data communication per query, AVG_COST, on the network. AVG_COST includes the cost of the semi-join program an algorithm produces and the cost of the final transmissions of relations to the result node.

In the figures that follow, some quantity is always plotted

against the number of relations or the number of attributes or the average selectivity. When the number of relations is stepwise varied, say, the number of attributes is varied randomly in a uniform fashion over the values 1 to 5 and the average selectivity is varied randomly in a uniform fashion over the range 0 to 1. For each number of relations many random queries are generated and tested so that a steady value of AVG_COST for these experimental conditions is determined. Likewise, the number of attributes and the average selectivity are stepwise varied. In this way we can get a feeling for how each parameter (relations, attributes and average selectivity) affects the performance of the algorithms tested. Each parameter is tested for queries with a minimum, an intermediate and a maximum number of join clauses, respectively.

### 6.5.1 IMPORTANCE OF THE INITIAL FEASIBLE SOLUTION

The IFS was determined to be the best query processing strategy only 0.13% of the time for the total number of queries tested by algorithm BLACK. When broken down in terms of the number of join causes, the IFS was determined to be the best query processing strategy 0.02% of the time for queries with a minimum number of join clauses, 0.05% of the time for queries with an intermediate number of join clauses and 0.06% of the time for queries with a maximum number of join clauses.

Clearly the IFS does not play a significant role as a query processing strategy when good semi-join preprocessing strategies are available.

### 6.5.2 COST COMPARISON OF THE IFS TO SEMI-JOIN PREPROCESSING STRATEGIES

For each query, the cost in data communication of the IFS is the sum of the relation sizes of the relations involved in the query. Algorithm BLACK only produces a semi-join program if it can improve upon the cost of the IFS. Algorithm WS further reduces this cost by rearranging or changing the semi-join program produced by algorithm BLACK. Since only 0.13% of all queries tested by algorithm BLACK had the IFS as the best query processing strategy, we can ignore the contribution of IFS costs in our results. If we compare the IFS cost for a query (IFS COST) to the actual cost of the query processing strategy produced (AVG_COST) by algorithm BLACK in conjunction with algorithm WS for that query, we get the results shown in Figures 5, 6 and 7.

In Figure 5 as the number of relations increases the ratio of the IFS COST to AVG_COST increases. For an intermediate number of join clauses when we have greater than 3 distributed relations in a query, the average improvement of the semi-join preprocessing strategies over the IFS is greater than 10:1. The average improvement is never less than 5:1.

In Figure 6, as we increase the number of attributes in queries, the ratio of the IFS COST to AVG_COST also increases. For an intermediate number of join clauses this improvement is never less than 11:1 and the improvement is approximately linear as the number of attributes increases.

In Figure 7 as the average selectivity increases, the ratio

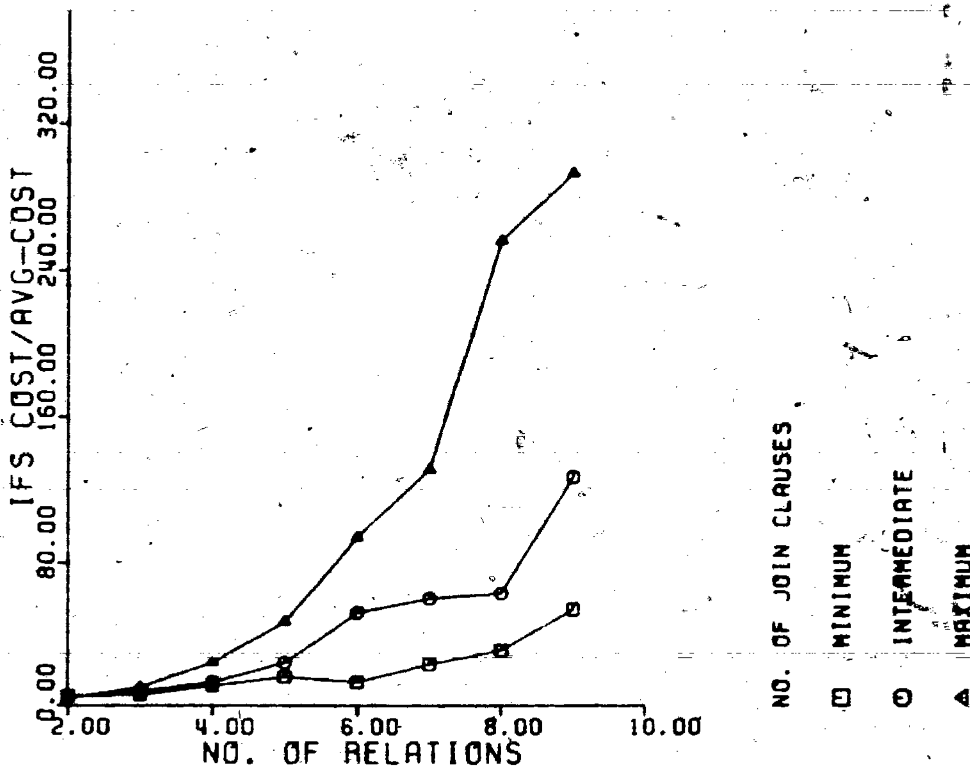



FIGURE 5:

IFS COST/AVG_COST versus No. of Relations

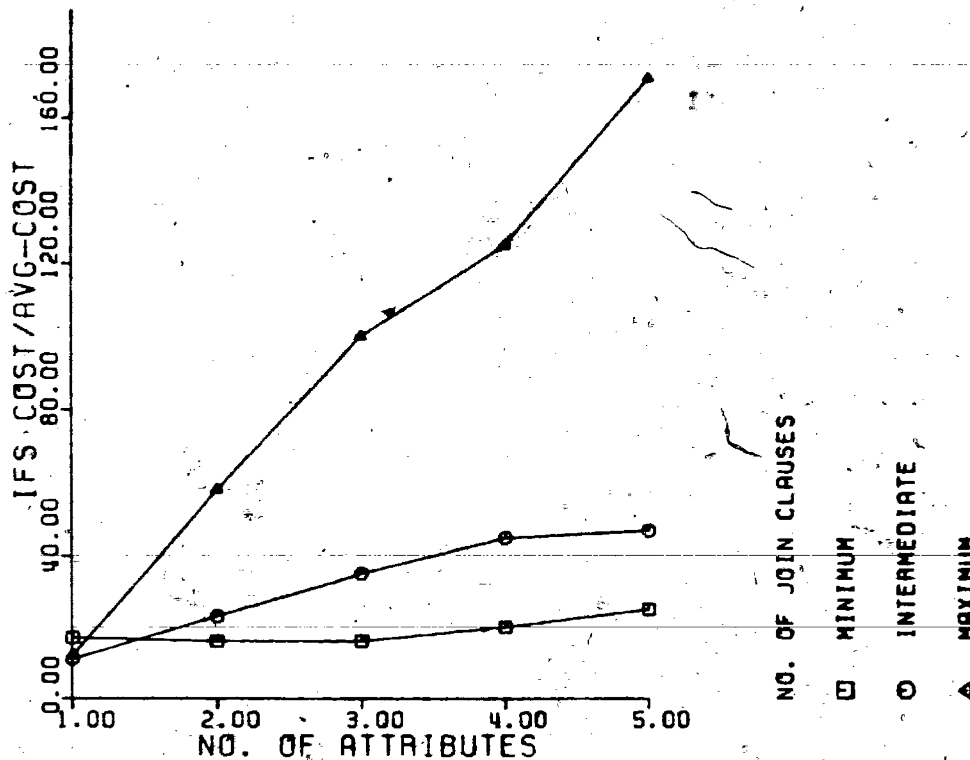



FIGURE 6:

IFS COST/AVG_COST versus No. of Attributes

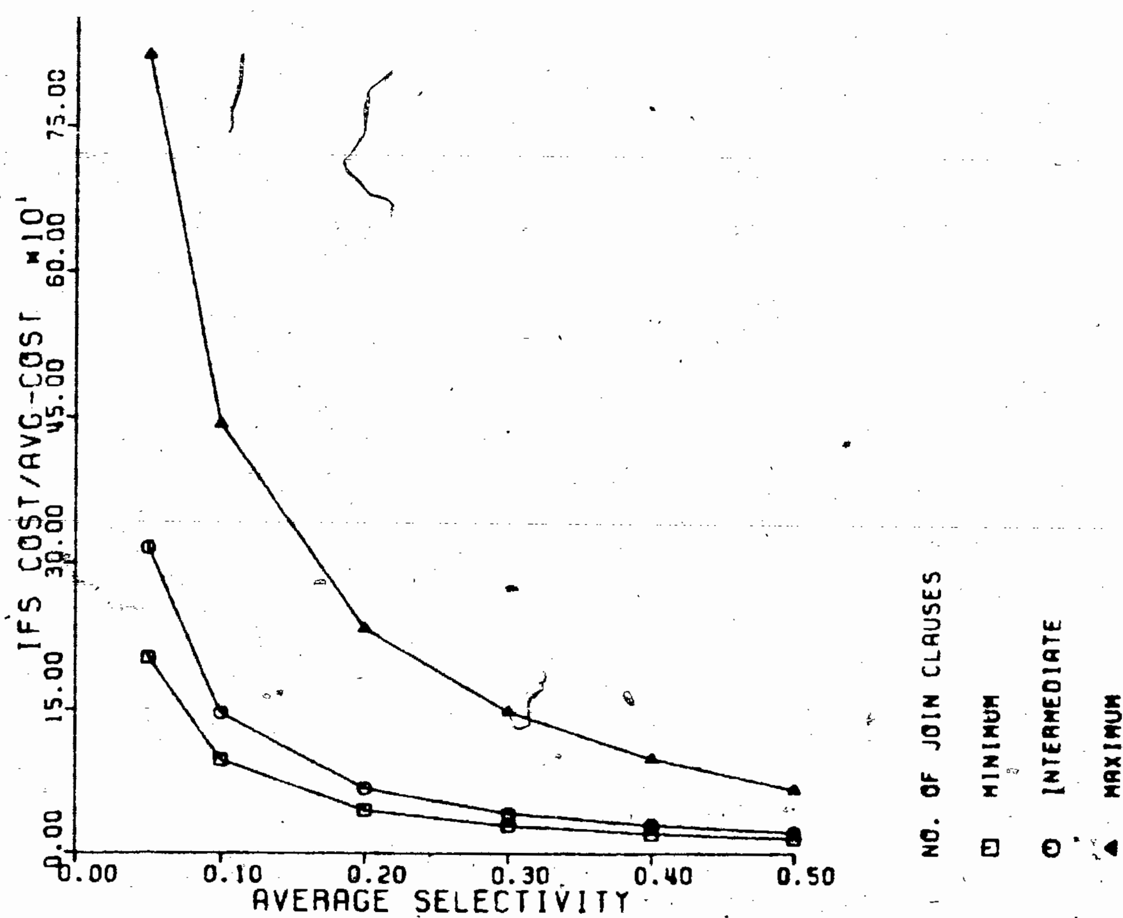



FIGURE 7:

IFS COST/AVG_COST versus Average Selectivity

of the IFS COST to AVG_COST decreases. This is as one would expect, except that this ratio decreases exponentially. This fact indicates that proper selectivity estimation is a critical factor if semi-join preprocessing strategies will perform in reality as they do on one's DDSM. For an intermediate number of join clauses the improvement of semi-join tactics over the IFS is never less than 22:1.

### 6.5.3 IMPORTANCE OF THE NUMBER OF SEMI-JOINS

There is some question as to whether the number of semi-joins in a semi-join preprocessing strategy is important. In a packet switched network it is the total amount of data to be transmitted that is significant. In this type of network the number of semi-joins to be executed should be unimportant. In an ordinary switched network the start-up cost causes the number of semi-joins to be executed to become sigificant [Hevn79a]. Whatever type of network is utilized, there is for each semi-join some overhead to consider in local processing by the local distributed database management system. This overhead consists of the number of instructions required to generate the information header for transmission of the semi-join operation on the network. This cost has been assumed to be zero in this work, but if preprocessing strategies are produced with a large number of semi-joins this cost could become a significant factor.

To get a feel for how the number of semi-joins varies in preprocessing strategies, we compiled data on the average number

of semi-joins produced in preprocessing strategies for algorithm BLACK for all types of queries tested. Algorithm BLACK on average always produced fewer semi-joins in the query preprocessing strategies than the other algorithms.

In Figure 8 as the number of relations increases the number of semi-joins produced in the preprocessing strategies increases linearly. The number of semi-joins produced when the the number of attributes or the average selectivity is varied, remains constant. We conclude that it is solely the number of relations (or nodes in the distributed database) referenced in a query that determines, on average, the number of semi-joins produced in preprocessing strategies generated by algorithm BLACK. This is also true for algorithm AP. The interesting thing to note from Figure 8 is that the number of join clauses in a query (i.e. the query complexity) has little to do with the number of semi-joins produced in the preprocessing strategies. This result is indirectly supported by the fact that varying the number of attributes produced no changes in the number of semi-joins in the preprocessing strategies. The rationale for this result is difficult to assess. Algorithms AP and BLACK always choose the next least cost semi-join to be added to the preprocessing strategy. A least cost semi-join is effectively the one with the highest selectivity (the domain sizes being equal). Relation cardinalities will decrease rapidly as semi-joins tend toward even higher selectivities. Our result says that slightly less than 2 semi-joins are required to reduce each relation enough so that there will be no more cost beneficial semi-joins. Even when

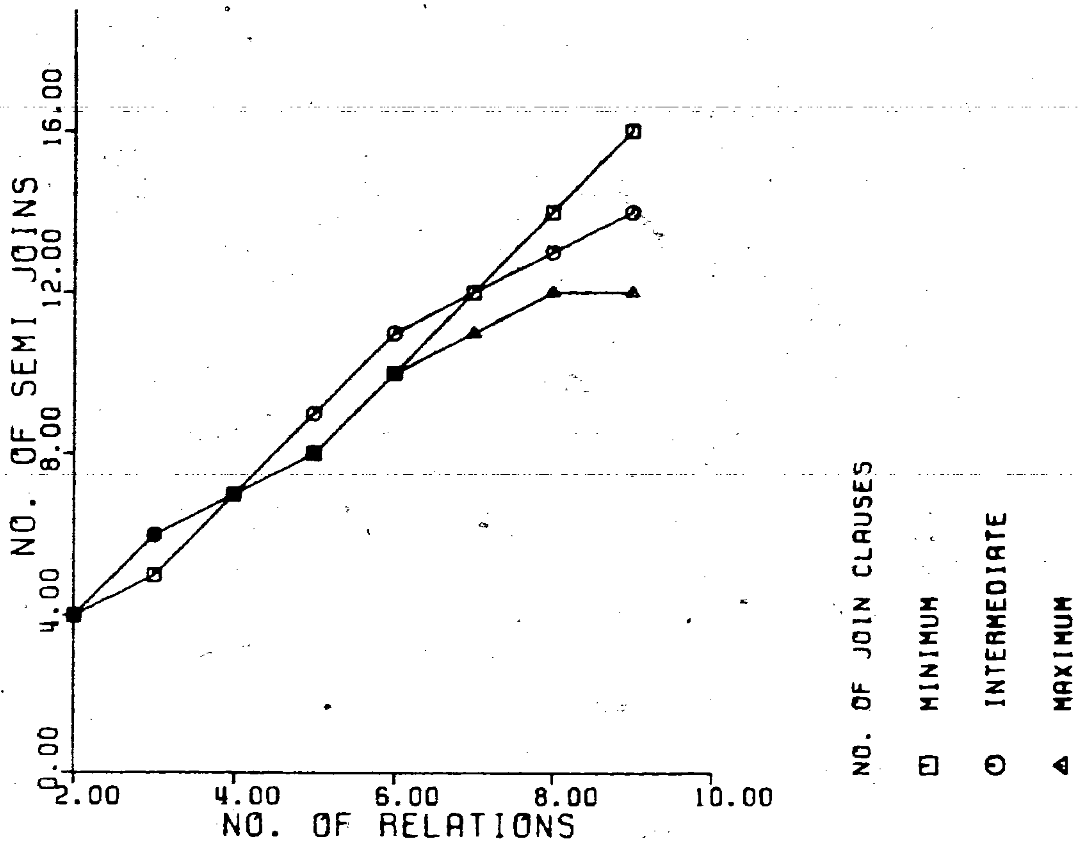



FIGURE 8:

No. of Semi-Joins versus No. of Relations

there are a minimum number of join clauses this result holds.

Considering the large decrease in network communication costs produced by semi-join preprocessing over the IFS and the relatively small number of semi-joins (2 per referenced node on average) in these strategies, the semi-join tactic must be considered a very good one.

6.5.4 SEMI-JOIN PROGRAM COST

When a semi-join preprocessing strategy has been derived, what proportion of AVG_COST is the semi-join program itself? AVG_COST is made up of the cost of the semi-join program and the cost of the final transmission of the reduced relations to the result node. The cost of the semi-join programs cannot be isolated independently from AVG_COST. The cheaper a semi-join program is, the higher its overall effective selectivity on the database will be; hence, the final size of the distributed relations will also be smaller.

Figures 9, 10 and 11 present the results obtained when the number of relations, the number of attributes and the average selectivity are varied, respectively. Figure 9 shows that the proportion of the cost of the semi-join program in AVG_COST generally increases as the number of relations increases. There is a fair amount of scatter in these results. The reason for this is not known. The same result applies in Figure 10 as the number of attributes in a query is varied. In Figure 11, as the average selectivity is varied, the semi-join program cost is in constant proportion to AVG_COST, except when there is a maximum

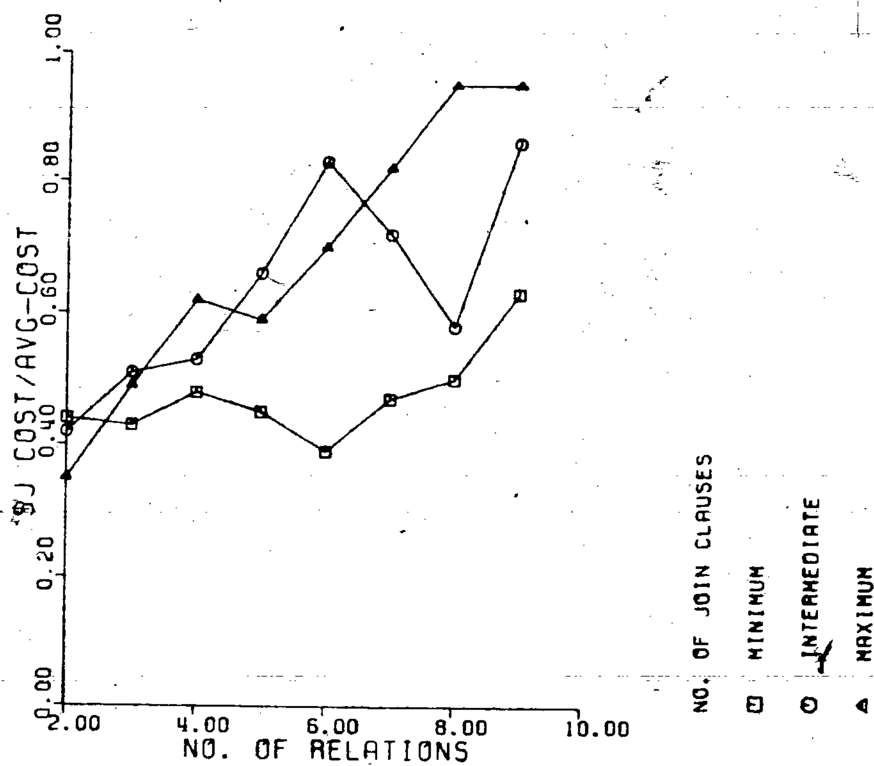



FIGURE 9:

SJ COST/AVG_COST versus No. of Relations

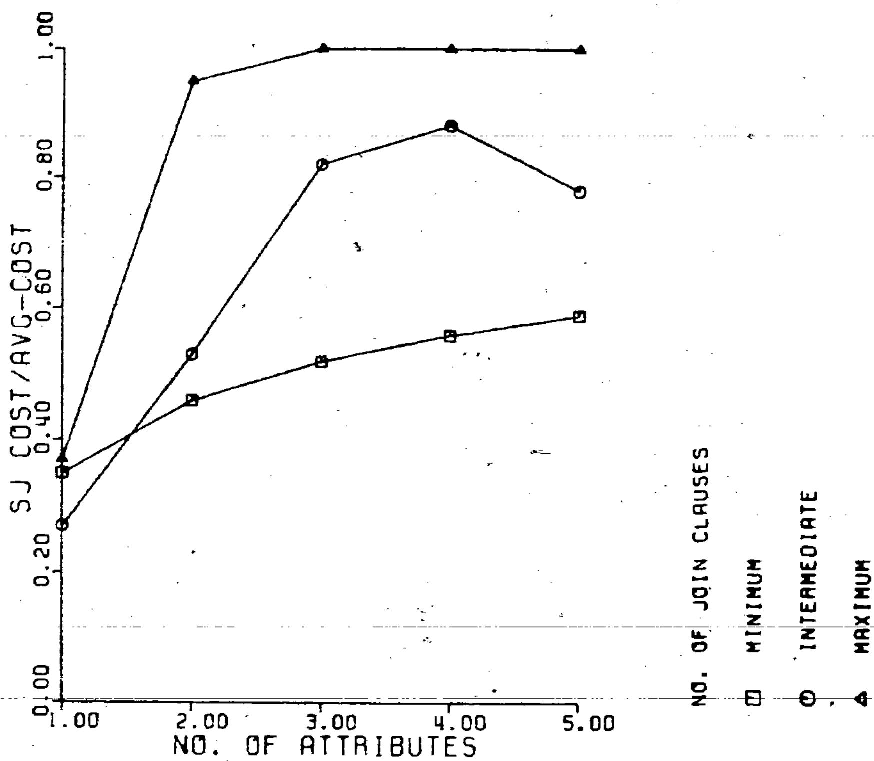



FIGURE 10:

SJ COST/AVG_COST versus No. of Attributes

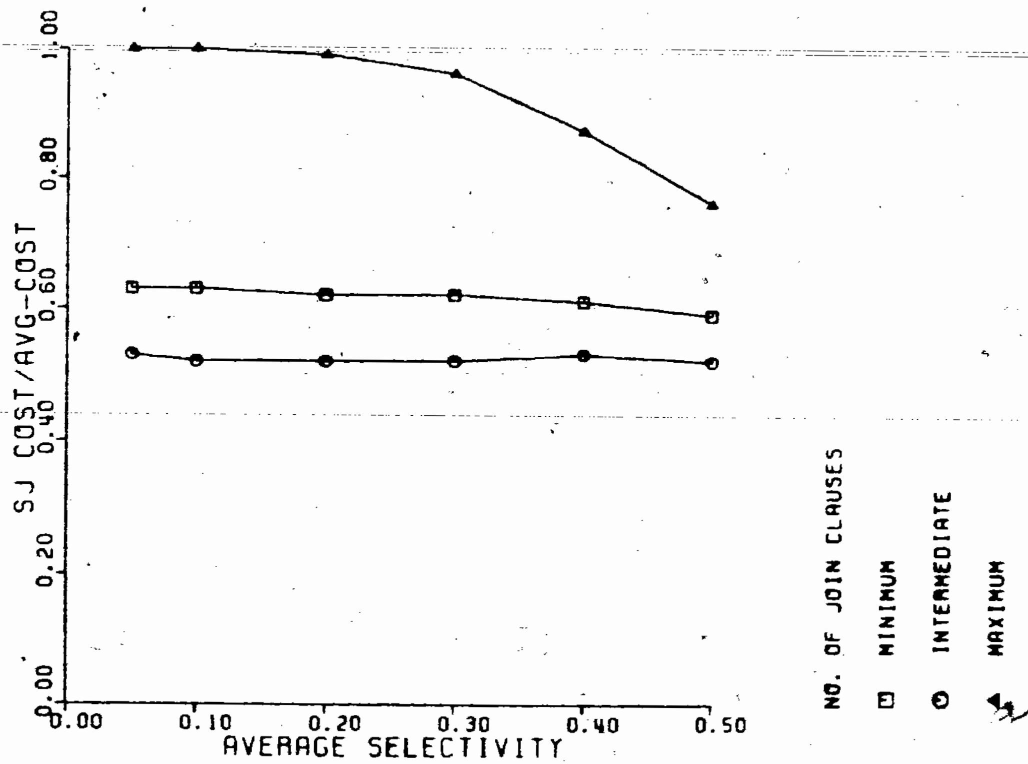



FIGURE 11:

SJ COST/AVG_COST versus Average Selectivity

number of join clauses.

The low percentage of the semi-join program cost in AVG_COST when we have few numbers of relations or attributes indicates a better semi-join program could produce more significant improvements in AVG_COST. Because under these conditions there are fewer semi-joins to choose from initially, some very good new heuristics will be needed to produce these improvements.

### 6.5.5 COMPARISON OF ALGORITHMS

The performance of algorithms AP, BLACK, HEVNER and OPT in formulating query processing strategies is now investigated. Since the cost of the IFS varies for each query, all the costs of query processing strategies (AVG_COST's) have been normalized so that the cost of the IFS for each query is 1000. This will allow the comparison of results between the algorithms. AVG_COST then has a maximum value of 1000. All semi-join preprocessing strategies formulated by algorithms AP, HEVNER and OPT are re-cost under our DDSM so that their performance may be compared with algorithm BLACK (and BLACK+WS).

In the next section, the effect of varying the number of relations in a query is investigated.

#### 6.5.5.1 RESULTS FROM VARYING THE NUMBER OF RELATIONS

In Figures 12 to 14, AVG_COST is plotted against the number of relations for queries with a minimum, an intermediate and a maximum number of join clauses, respectively. Five curves are

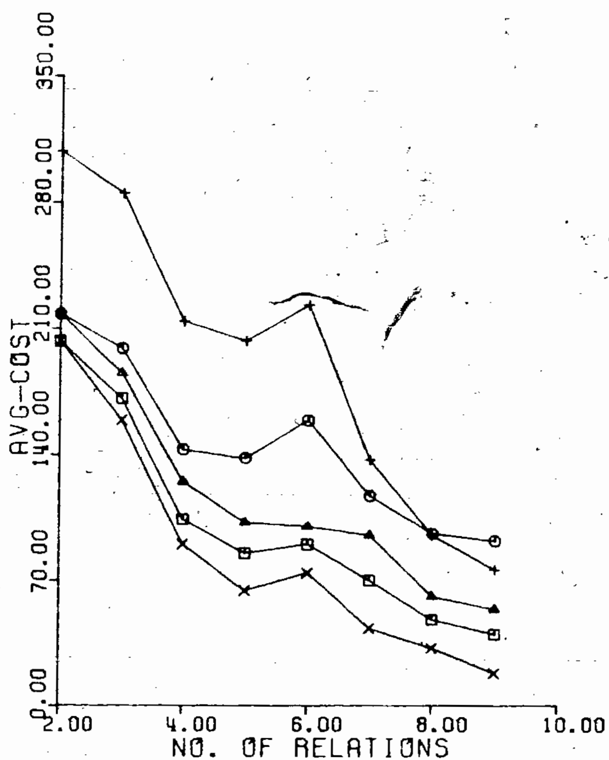



FIGURE 12:

AVG_COST versus No. of Relations

for a Minimum No. of Join Clauses

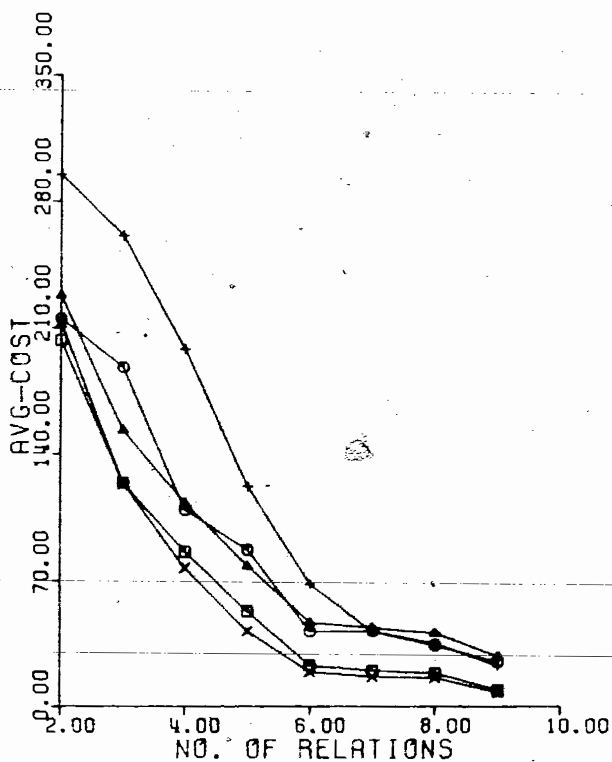



FIGURE 13:

AVG_COST versus No. of Relations

for an Intermediate No. of Join Clauses

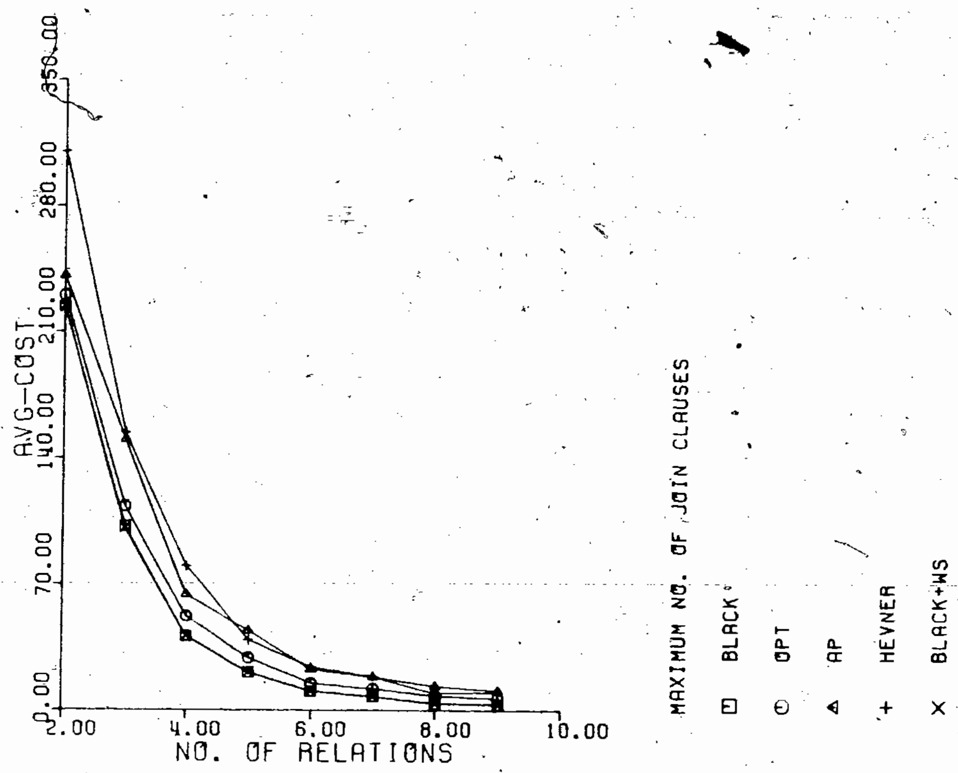



FIGURE 14:

AVG_COST versus No. of Relations for a maximum No. of Join Clauses

present in each figure, one for each algorithm tested.

In the figures, as the number of relations increases, the cost of the semi-join preprocessing strategies decreases as a percentage of the cost of the IFS. This trend is also noticed as the number of join clauses in a query increases. We can conclude that when there are more join clauses in a query (hence more semi-joins to choose from) query preprocessing strategies formulated by these algorithms become cheaper.

Table 1, presented below, summarizes the results in Figures 12 to 14. The values quoted are the average value of AVG_COST for the points on each curve.

| ALGORITHM | NUMBER OF JOIN CLAUSES | | |
|---|---|---|---|
| | MINIMUM | INTERMEDIATE | MAXIMUM |
| BLACK | 101 | 67 | 52 |
| OPT | 145 | 93 | 58 |
| AP | 118 | 92 | 71 |
| HEVNER | 193 | 131 | 81 |
| BLACK+WS | 85 | 64 | 52 |

Table 1:
Average AVG_COST when varying
the No. of Relations

From the results in Table 1, we see that algorithm HEVNER performs worse, on average, than any of the other algorithms. It is interesting to note that the new version of algorithm AP, OPT, performs worse than AP when there are a minimum and intermediate number of join clauses in queries. Since OPT has incorporated a solution for the semi-join history problem, one would expect it to perform better than AP, which does not

utilize this result. The OPT heurisitic of choosing semi-joins with a maximum profit must be a very poor strategy, indeed. This also indicates that the heuristic used in AP; choosing the least cost semi-join, is very good. Algorithm BLACK uses an extension of AP's heuristic and also incorpotates a solution for the semi-join history problem. One would expect BLACK to perform better than AP or OPT, which it does, significantly. The 'speed-up' algorithm WS provides on the semi-join programs produced by BLACK gives an even greater improvement.

6.5.5.2 RESULTS FROM VARYING THE NUMBER OF ATTRIBUTES

In Figures 15 to 17, AVG_COST is plotted against the number of attributes for queries with a minimum, an intermediate and a maximum number of join clauses, respectively.

For queries with a minimum number of join clauses, as the number of attributes increases, the cost of the semi-join preprocessing strategies produced by HEVNER and OPT increases.

Since there are a minimum number of join clauses in these queries (i.e. a minimal number of edges in the query graph), each relation schedule HEVNER produces will contain fewer and fewer semi-joins as the number of attributes increases. Thus the effective selectivity of each schedule will decrease and, as seen, AVG_COST increases as the number of attributes increases. With only one attribute, the queries are basically simple queries and the relation schedules formed will have a maximum number of semi-joins present. HEVNER performs best in this case. As the number of join clauses in queries increases, this case

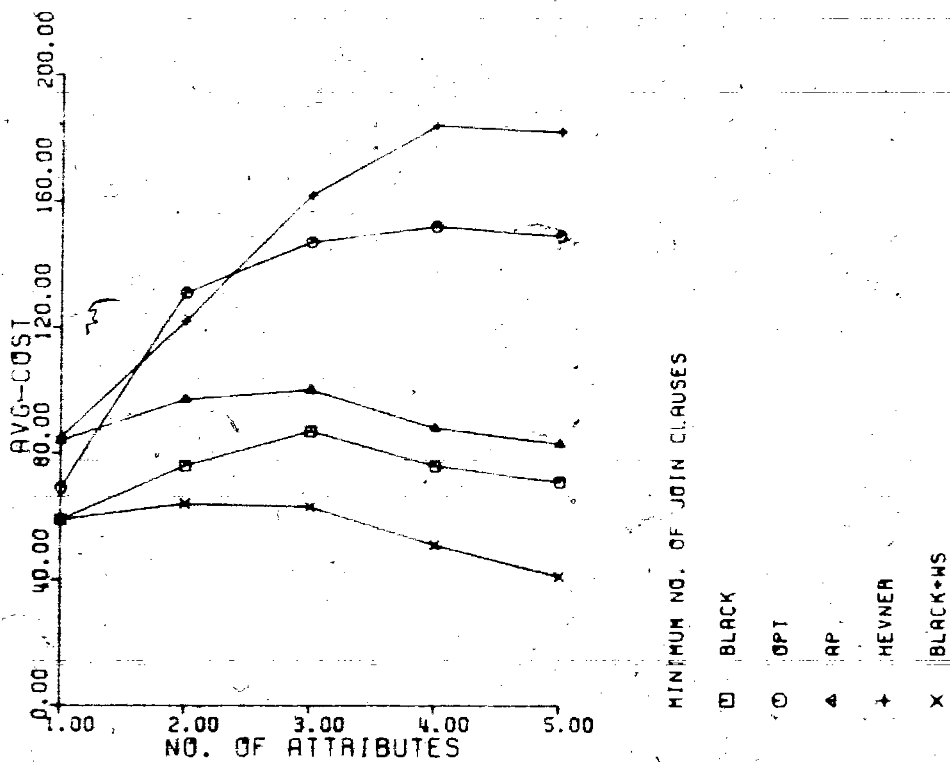



FIGURE 15:

AVG_COST versus No. of Attributes
for a Minimum No. of Join Clauses

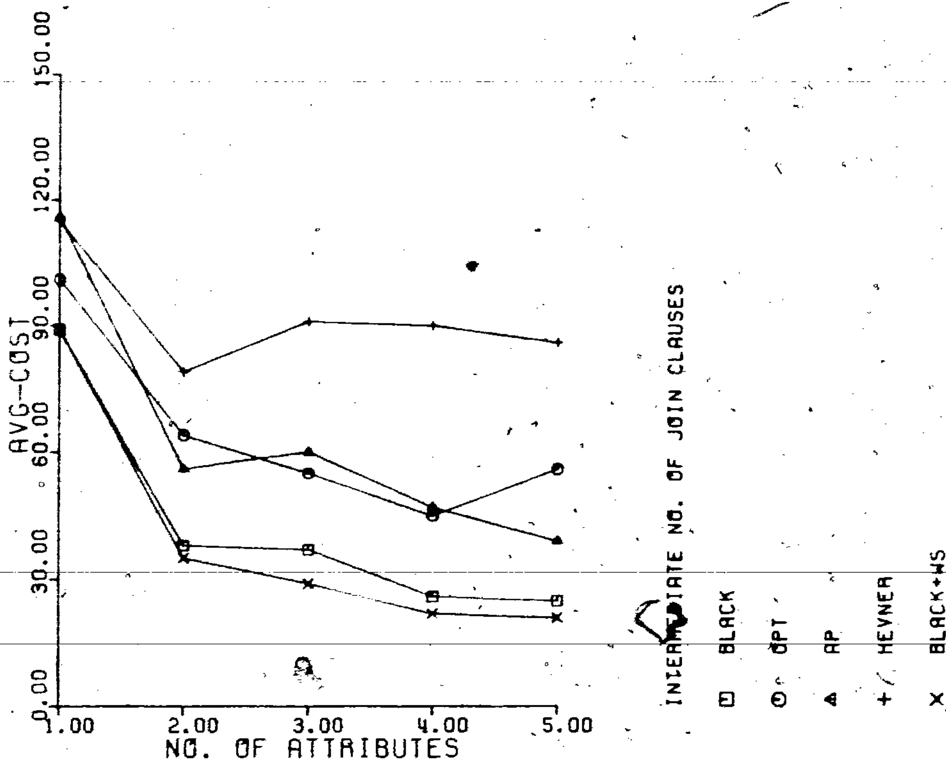



FIGURE 16:

AVG_COST versus No. of Attributes
for an Intermediate No. of Join Clauses

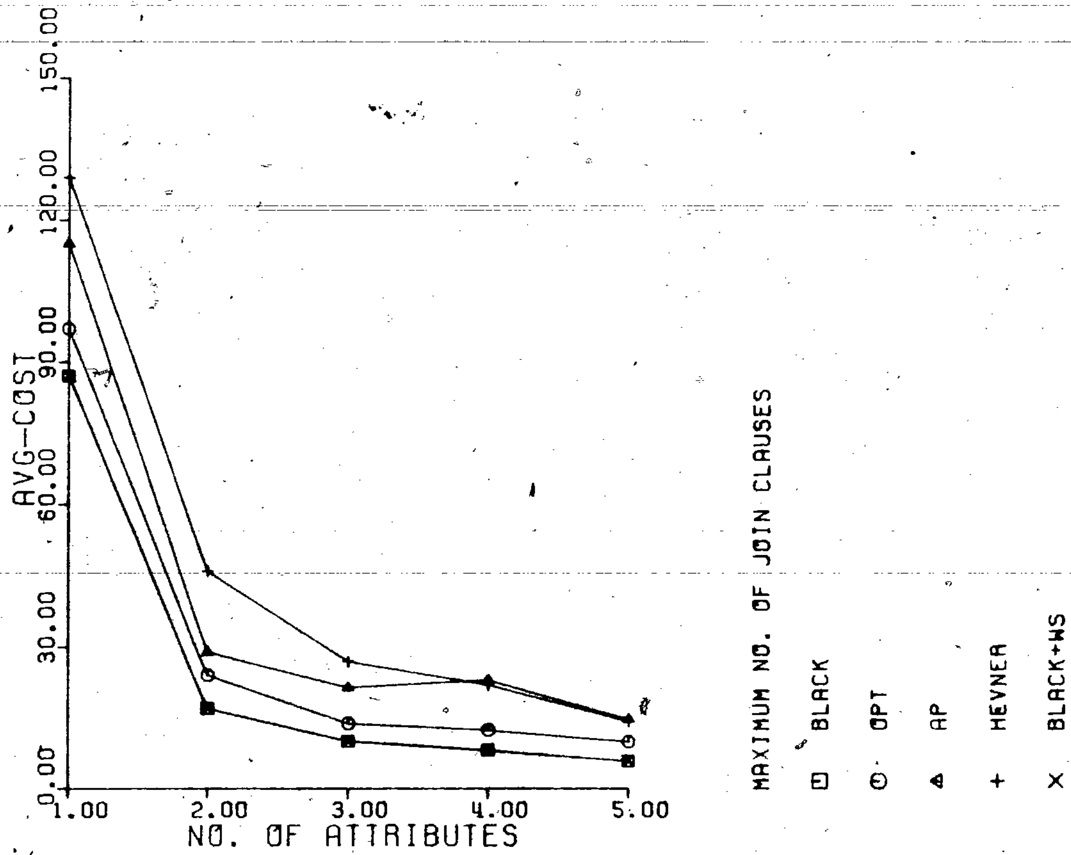



FIGURE 17:

AVG_COST versus No. of Attributes for a Maximum No. of Join Clauses

becomes less favorable, since relation schedules will become more cost beneficial with parallel attribute schedules, thus increasing the overal selectivity of the schedule.

The behaviour of OPT is very similar to that of HEVNER. For a minimum number of join clauses, when we have one attribute in the queries, OPT has the most candidate semi-joins to choose from. As the number of attributes increases, the number of join clauses remains constant, but the number of implied semi-joins decreases. One can conclude that the heuristic used in OPT for choosing semi-joins does not work well when it choice of profitable semi-joins is limited.

Table 2, presented below, summarizes the results shown in Figures 15 to 17.

| ALGORITHM | NUMBER OF JOIN CLAUSES | | |
|---|---|---|---|
| | MINIMUM | INTERMEDIATE | MAXIMUM |
| BLACK | 74 | 43 | 26 |
| OPT | 130 | 64 | 31 |
| AP | 90 | 64 | 41 |
| HEVNER | 147 | 92 | 48 |
| BLACK+WS | 56 | 39 | 26 |

Table 2:
Average AVG_COST when varying
the No. of Attributes

The results presented in Table 2 show the same trends as the results presented in Table 1 where the number of relations in queries was varied.

6.5.5.3 RESULTS FROM VARYING THE AVERAGE SELECTIVITY

In Figures 18 to 20, AVG_COST is plotted against the average selectivity for queries with a minimum, an intermediate and a maximum number of join clauses, respectively.

For each algorithm as the average selectivity is varied from 0.05 to 0.50, AVG_COST increases. Semi-joins with an initial average selectivity of 0.05 would be expected to have greater reducing effect on the size of the database than semi-joins with an average selectivity of 0.50.

For a particular selectivity value, as the number of join clauses in queries increases, AVG_COST decreases. When there are more join clauses in queries, this result reflects the greater choice and greater numbers of semi-joins to choose from when forming query preprocessing strategies.

Table 3, presented below, summarizes the results in Figures 18 to 20.

| ALGORITHM | NUMBER OF JOIN CLAUSES | | |
|---|---|---|---|
| | MINIMUM | INTERMEDIATE | MAXIMUM |
| BLACK | 44 | 23 | 7 |
| OPT | 75 | 36 | 10 |
| AP | 48 | 36 | 15 |
| HEVNER | 86 | 44 | 16 |
| BLACK+WS | 30 | 21 | 7 |

Table 3:
Average AVG_COST when varying the Average Selectivity

Again, the same general trend of results is noticed as in Tables 1 and 2.

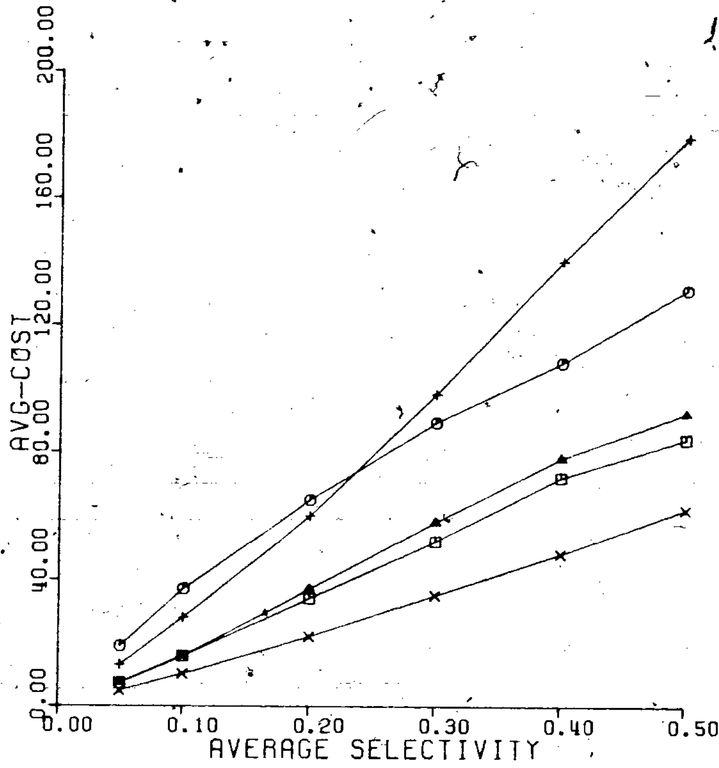



FIGURE 18:

AVG_COST versus Average Selectivity

for a Minimum No. of Join Clauses

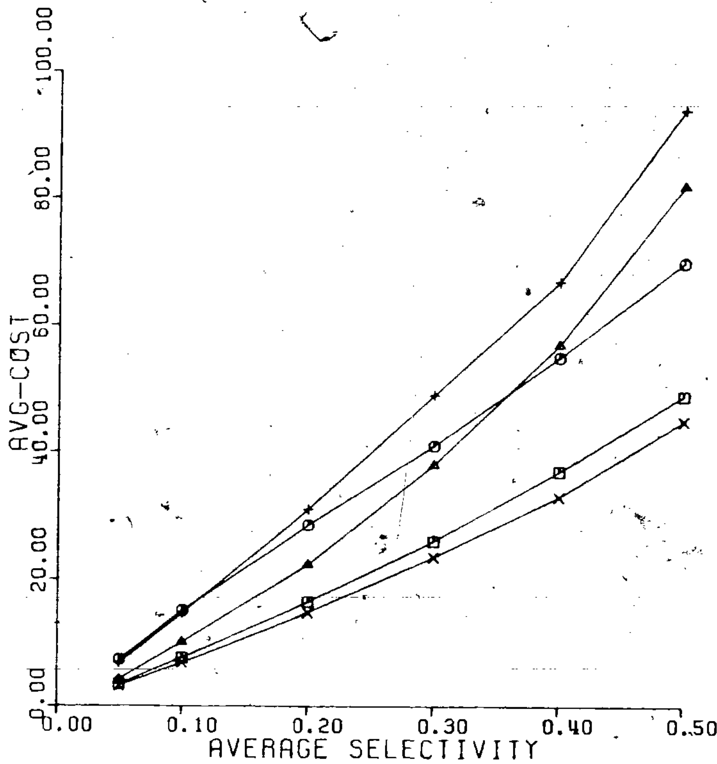



FIGURE 19:

AVG_COST versus Average Selectivity

for an Intermediate No. of Join Clauses

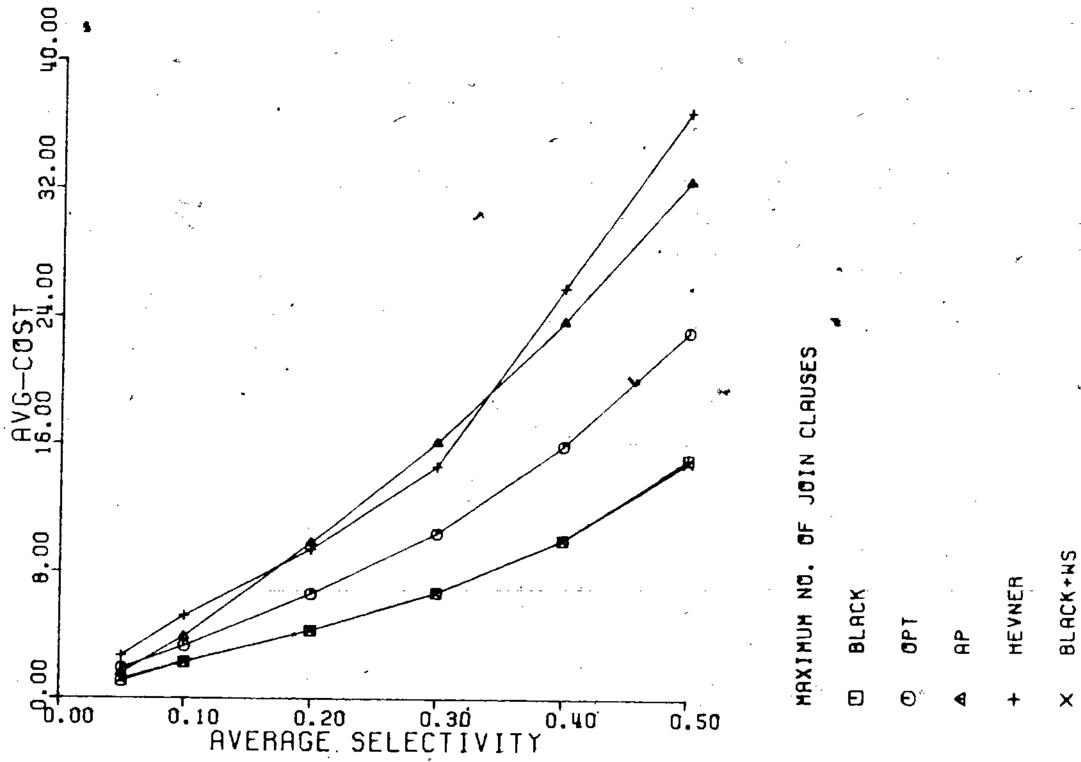



FIGURE 20:

AVG_COST versus Average Selectivity for a Maximum No. of Join Clauses

6.5.5.4 SUMMARY OF RESULTS

No one of the algorithms has any time advantage in deriving a query preprocessing strategy for a query, since they all have the same order of time complexity. (Algorithm BLACK will derive a strategy for a query in the order of a few milliseconds of computer time.)

Algorithm HEVNER always performs worse than the other algorithms. The strategies it derives are at least twice as costly as those derived by algorithm BLACK.

Algorithms AP and OPT peform similarly when queries have an intermediate number of join clauses. AP is significantly better than OPT when queries have a minimum number of join clauses. OPT is slightly better than AP when queries have a maximum number of join clauses.

Algorithm WS produces its greatest improvements on the output of algorithm BLACK when queries have a minimum number of join clauses. When queries have a maximum number of join clauses algorithm WS produces little improvement.

# 7. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

We have seen that the DDSM's that algorithms AP and HEVNER utilize are not realistic enough since the semi-join preprocessing strategies produced do not perform well under our DDSM.

The parallel independent relation schedules that algorithm HEVNER produces are the main reason for its failure to perform well when compared to our algorithm.

Ignoring the semi-join history problem is the main failing of algorithm AP's method of producing semi-join preprocessing strategies.

We saw that recent improvements in algorithm AP (OPT) have not produced a better algorithm. Choosing the most profitable semi-join for inclusion in RHO is not a good heuristic and it cancels the benefit OPT has over AP in incorporating a solution for the semi-join history problem.

Algorithm BLACK is significantly better than algorithms AP and OPT in producing good semi-join preprocessing strategies. Algorithm HEVNER, even with our enhancement, does not produce good semi-join preprocessing strategies. It is unlikely that HEVNER can be improved significantly with a minor modification of heuristics. Algorithm WS achieves further significant reduction of query processing costs by 'speeding-up' the semi-join preprocessing strategies formulated by algorithm BLACK.

One question concerning this research that has not been answered is, how close are our results to an optimal solution?

At the present time no worker has proposed an algorithm for the optimal solution. If the results presented here are within a factor of 2 to the optimal solution, more work in this area is unnecessary. Even if an optimal solution were proposed, since the algorithm will necessarily be exponential, it may be possible to determine optimal results only for a very limited class of queries. This question needs to be answered to provide a benchmark for heuristic algorithms. Some novel solution will be necessary to allow such an algorithm to determine results for the general class of queries.

# REFERENCES

Astr76
Astrahan, M.M., et al, System R: A Relational Approach to Data Management, ACM Transactions on Database Systems, 1:2, 1976, pp. 97-137.

Bern81a
Bernstein, P.A., and D.W. Chui, Using Semi-Joins to Solve Relational Queries, JACM 28, 1 (Jan. 1981), pp 25-40.

Bern81b
Bernstein, P.A., and N. Goodman, The Power of Natural Semi-Joins, SIAM J. Comput. 10, 4 (Nov. 1981).

Chiu80
Chiu, D.M., and Y.C. Ho, A Methodology for Interpreting Tree Queries into Optimal Semi-Join Expressions, In Proc. ACM SIGMOD Conf. May 1980.

CODA71
CODASYL [1971], CODASYL Data Base Task Group April 1971 Report, ACM, New York.

Codd70
Codd, E.F., A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

Codd71
Codd, E.F., Normalized Database Structure: A Brief Tutorial, Proceedings ACM SIGFIDET Workshop, New York, 1971.

Codd72
Codd, E.F., Relational Completeness of Data Base Sublanguages, in Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall, 1972, pp. 65-98.

Good79
Goodman, N., P.A. Bernstein, E. Wong, C.L. Reeve, J.B. Rothnie, Query Processing in SDD-1: A System for Distributed Databases, TR CCA-79-06, Oct. 1979, Computer Corporation of America.

Good81
Goodman, N., P.A. Bernstein, E. Wong, C.L. Reeve, J.B. Rothnie, Query Processing in a System for Distributed Databases (SDD-1), ACM Transactions on Database Systems, Vol. 6, No. 4, Dec. 1981, pp. 602-625.

Hevn79a
Hevner, A.R., Optimization of Query Processing in Distributed Database Systems, Ph.D. Thesis, Purdue University, 1979.

Hevn79b
Hevner, A.R., S.B. Yao, Query Processing in Distributed Database Systems, IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979.

IBM78
IBM [1978], IMS/VS publications GH20-1260, SH20-9025, SH20-9026 and SH20-9027, IBM, White Plains, N.Y.

Luk80
Luk, W.S., Optimizing Query Processing in a Distributed Database System, TR 80-4, 1980, Simon Fraser University.

Luk81
Luk, W.S., On Estimating Block Accesses in Database Organizations, TR 81-10, 1981, Simon Fraser University.

Mart76
Martin, J., Telecommunication and the Computer, Prentice-Hall, Englewood Cliffs, 2nd edition, 1976.

Ston76
Stonebraker, M., E. Wong, P. Kreps, and G. Held, The Design and Implementation of INGRES, ACM Transactions on Database Systems, Vol. 1, No. 3, Sept. 1976, pp. 189-222.

Wong79
Wong, E., Retrieving Dispersed Data From SDD-1: A System for Distributed Databases, Proc. 1977 Berkeley Workshop on Dist. Data Man. and Comp. Networks, May 1979.

Yao77
Yao, S.B., Approximating Block Accesses to Database Organizations,CACM 20, 1977.

Yu79
Yu, C.T., K. Lam and M. Ozsoyoglu, An algorithm for Tree-Query Membership of a Distributed Query, In Proc. COMPSAC 1979, IEEE Comp. Society, November 1979.

Zani79
Zaniolo, C., Design of Relational Views over Network Schemas, International Conference on the Management of Data, ACM-SIGMOD 1979, pp. 179-190.

Zipf49
Zipf, G.K., Human Behaviour and the Principle of Least Effort, Addison-Wesley, Cambridge Mass., 1949.