DISTRIBUTED QUERY PROCESSING IN LOCAL NETWORK DATABASES

Thomas Page

1984
Report No. CSD-840222

MASTER COPY

SECURE DISTRIBUTED PROCESSING SYSTEMS

QUARTERLY TECHNICAL REPORT

October 1983 - December 1983

Gerald J. Popek
Principal Investigator
Computer Science Department
School of Engineering and Applied Science
University of California at Los Angeles
(213) 825-6507

This research was sponsored by the
Defense Advanced Research Projects Agency

ARPA Contract No.: MDA-903-82-C-0189
ARPA Order No.: 3396
Program Code No.: 7P10

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Query Processing in Local Network Databases

A thesis submitted in partial satisfaction of the requirement for the degree of Master of Science in Computer Science

by

Thomas Wingfield Page, Jr.

1983

Table of Contents

page

1 Introduction 1
1.1 Introduction to Distributed Databases 1
1.2 Introduction to Query Processing 3
1.3 Purpose of the Thesis 7
1.4 Thesis Plan 11

2 Distributed Database Model 13
2.1 The Network Model 13
2.2 The Database Model 14
2.2.1 Horizontal Fragments 14
2.2.2 Replicated Copies 16
2.3 Database Cost Model 16

3 Query Processing in Distributed Databases 19
3.1 Introduction 19
3.2 Searching the Solution Space 21
3.3 The Distributed Ingres Proposal 24
3.3.1 Query Processing in Single Site Ingres 25
3.3.2 Query Processing in Berkeley Distributed Ingres 32
3.3.2.1 The Distributed Ingres Algorithm 33
3.3.2.2 Processing Cost Function 37
3.3.2.3 The Communications Cost Function 38
3.3.2.4 Minimizing Communications Costs 39
3.3.2.5 Minimizing Processing Costs 40
3.3.2.6 Running the Query 40
3.3.3 Discussion of the Ingres Algorithm 41
3.4 Query Processing in SDD-1 46
3.4.1 The Hill Climbing Approach 46
3.4.2 The Final Version 48
3.5 Distributed Query Compilation in R* 50

4 The Local Network Environment 54
4.1 The Computing Environment of the Future 54
4.2 How This Differs From Modern Networks 56
4.2.1 Qualitative Differences 56
4.2.2 Quantitative Differences 57
4.3 Implications for Query Processing 61

5 Case Study, A Distributed Database in LOCUS 65
5.1 Conventional DDMS Architecture 65
5.2 Two New Architectures 68
5.3 Implementation 70
5.4 Performance Results 76
5.5 Comparison With Berkeley Distributed Ingres 79

6 Conclusions and Future Research 83

6.1 Query Processing 83
6.2 Distributed Databases in LOCUS 85
6.3 Future Research 86

Appendix 1 Determining Cost Ratios 88

References 93

List of Figures

page

Figure 1: Ingres Query Tree. 27

Figure 2: Ingres Query Decomposition Calling Sequence. 31

Figure 3: Query processing in a distributed database system 67

Figure 4: Structure of the remote subquery mechanism 71

Table 1: Performance of different Ingres architectures 77

Table 2: Disk Costs 88

Table 3: Communications Costs 89

Table 4: Processing time versus number of clauses 90

Table 5: Processing time versus length of tuple 90

Table 6: Processing time versus number of attributes 90

Table 7: Cost per second of using resources 91

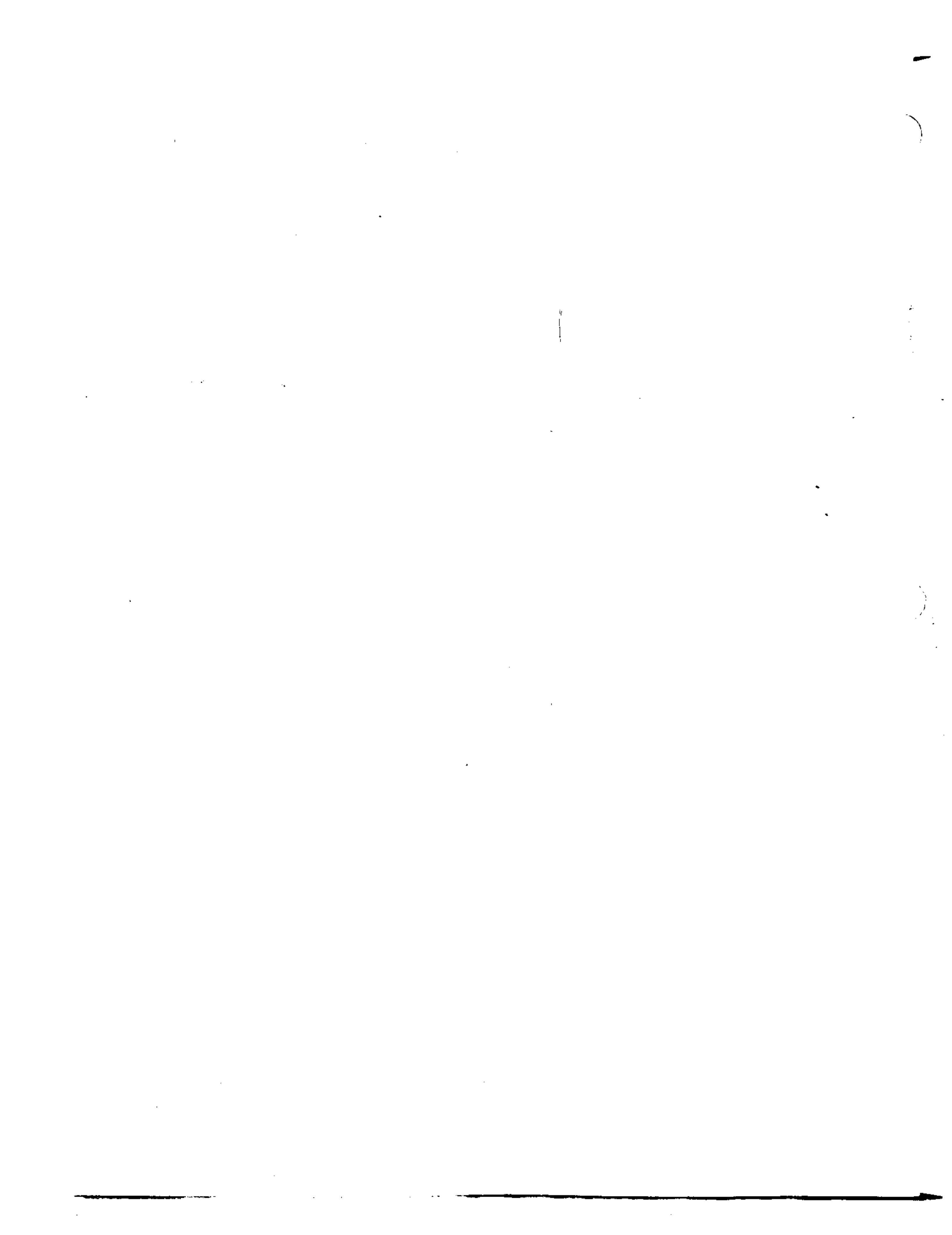
Table 8: Time per database operation on each resource 91

ACKNOWLEDGEMENTS

First, I must thank my advisor, Dr. Gerald Popek for his advice, encouragement and direction. His help was invaluable in the design and optimization of this work and his careful review of this thesis forced me to think through each point.

I would like to thank Dr. Stott Parker and Dr. Wesley Chu for being on my committee. Their thoughtful reading improved the quality of the final document.

I am also indebted to the members of the LOCUS project both past and present for creating the environment which made this work possible and to Dr. Terry Gray and the staff of the Center For Experimental Computer Science for improving our environment and keeping the systems running.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): RELIABLE DISTRIBUTED PROCESSING SYSTEMS
5. TYPE OF REPORT & PERIOD COVERED: Quarterly Technical Report October 1983-December 1983
6. PERFORMING ORG. REPORT NUMBER: CSD-840222
7. AUTHOR(s): Gerald J. Popek, Principal Investigator
8. CONTRACT OR GRANT NUMBER(s): MDA-903-82-C-0189
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Dept; School of Engineering & Applied Science; Univ. of California at Los Angeles; Los Angeles, Calif 90024
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, Virginia 22209
12. REPORT DATE: February 21, 1984
13. NUMBER OF PAGES: 104
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research 1030 East Green Street Pasadena, California 91101
15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report): Approved for public release. Distribution unlimited
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
18. SUPPLEMENTARY NOTES
19. KEY WORDS (Continue on reverse side if necessary and identify by block number): This research is to provide empirical results about the nature of high speed networks as an environment in which to perform distributed query optimization and to implement a remote query processing mechanism for Ingres.
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT OF THE THESIS

Distributed Query Processing in

Local Network Databases

by

Thomas Wingfield Page, Jr.

Master of Science in Computer Science

University of California, Los Angeles, 1983

Professor Gerald J. Popek, Chair

There are as yet no fully operational distributed relational databases for local area networks. Consequently, all work in this area has been purely theoretical, based on experience in geographically distributed systems, or on simulation and modeling. Current trends toward decentralized computer systems and the need for database capabilities in office systems necessitate the production of distributed databases for high speed local networks in the very near future. This thesis provides some necessary results pertaining to the problems of query processing and database management system architectures in local area computer networks.

The first goal of this research is to provide empirical results about the nature of high speed networks as an environment in which to perform distributed query optimization. Query optimizers in geographically distributed databases where the network is a bottleneck, most commonly attempt to minimize communications traffic. This thesis shows that in local networks, it is reasonable for the optimizer to treat network com-

munications as free. The costs of disk retrieval and local processing dominate communications cost; and thus, like in single site systems, the goal is to minimize disk and processing costs. However, the optimization problem in distributed databases differs from that in single site systems because of the potential for parallel processing presented by the many processors in the network.

The second goal of this thesis is the implementation of a remote query processing mechanism for Ingres. This mechanism is operational in the LOCUS distributed operating system and allows a single site Ingres module to perform subqueries at remote sites. The purpose of this effort is to address the difficulty of constructing distributed databases. Such systems are very much more expensive to build than their single site ancestors because of the different and more complex interfaces to remote resources, the more complicated model of the underlying system, the richer set of failure modes with which the software must deal, and the need to mask these problems from the user. Our distributed database is obtained by running a single site Ingres system in LOCUS which provides the database with a network transparent view of the underlying system. The operating system handles many of the problems that make building distributed databases difficult. It provides access to remote data, transactions, distributed directory management, replication, partitioned operation and recovery from failures. With the implementation of a remote query processing mechanism, the database is able to make use of the many processors in the network to speed the query by operating on portions of the job in parallel. This thesis demonstrates that a distributed database may be implemented quickly and efficiently using the distributed environment provided by the LOCUS operating system.

CHAPTER 1

Introduction

1.1 Introduction to Distributed Databases

A great number of computer science papers in recent years have begun with some subset of the following observations.

1. Equivalent computing power can often be provided more cheaply and flexibly by a collection of small machines than by a single large one.

2. Software is more costly, less reliable, and more time consuming to develop than hardware.

3. Software for distributed systems is very much more difficult to write than software for single machines.

A result of the first observation is that small systems are proliferating. Because of the smaller amounts of money involved, the decision to buy a small machine can be made by low to mid level management, encouraging a decentralized organization of computing resources. Users themselves retain autonomy over their own machines and thus have considerable say over such things as when the machines are to be taken down or when new software is to be installed.

A problem with this decentralization is that while the processors themselves have become cheap enough that we can afford to dedicate one or more per person, other resources must still be shared. Laser printers, super computers, plotters, and communications lines are but a few examples. The decentralized control that we sought is in direct conflict with the desire to coordinate the sharing of expensive resources efficiently and fairly.

Data is the resource whose effective sharing is most critical. It has been observed that data management is what computers really do in the business world. Scientific computation, control, CAD/CAM, simulations, and document preparation uses are dwarfed by the database applications. Databases are an invaluable tool in the centralized environment because they free the user or programmer from the concerns of physical data storage and they allow different users to share the same data without conflict. For precisely the same reason they will be needed in the decentralized computer systems of the future where the problems of the physical storage of data and the coordination of unrelated users take on added dimensions of complexity. Distributed databases allow us to store information near the point where it is most often needed yet still permit global access to it. It is the goal of distributed databases to make the user view of data no more complex than in single machine facilities.

Distributed databases are a current research "hotspot". As usual the two driving forces behind this interest are economics and silicon. Rapid advances in local network technology and the advent of personal workstations has made the distribution of resources possible. The economics of small machines has made the distribution of resources necessary. It is not just hardware resources which have become dispersed but also software and data.

The management of dispersed data is not a new problem. Many applications such as airline reservation systems or automatic tellers by their very nature create and consume data in a distributed way. However, the problem of data management in a local area network is quite different from the problems encountered in geographically distributed networks. Perhaps the main difference is the user model of such systems. Local area networks connecting small machines are replacing mainframes as the primary computing tool for most users. Thus users expect the new environment to provide equivalent if not superior semantics and performance to the central facilities to which they have become accustomed. Because of the response characteristics of local networks, users can view the network as a single resource. The applications created for local area networks exhibit a high degree of cooperation among machines and high degree of sharing of resources. Distribution of data in local networks occurs not only because data is created in a distributed fashion but also to increase reliability by replicating data, to increase speed by allowing several processors to work on one problem in parallel, and to improve average response time by keeping data near the users who need it most often.

1.2 Introduction to Query Processing

Our distributed database is organized according to the relational model of data [Codd 70]. This model provides data independence, semantic completeness and a logical view of the database which frees the user from concerns about the physical storage of data. A relational database consists of a collection of *relations*, each of which may be thought of as a flat table. Each row in the table is an individual entry in the relation and is called a *tuple*. The columns in the table are called *attributes*. For example, each tuple in an employee relation might have a name, number, and a salary attribute. The user expresses his query not by an access plan but instead by the contents of some

of the attributes of the tuples being addressed. He uses a non-procedural interface language such as Quel [Held 75], SQL [Chamberlin 76] or QBE [Zloof 77] and the onus is on the system to determine an efficient access path.

Quel is the language most often used for interactive access to an Ingres database and will be used in this paper for examples. [Held 75] contains a complete presentation of the language. We will briefly introduce the subset of the language considered in this thesis.

Quel supports four basic commands: RETRIEVE, REPLACE, DELETE, and APPEND. This discussion will be restricted to the processing of retrieves as the other commands are implemented using a retrieve. Examples in this thesis will use the following sample database.

```
supplier (Sname, S#, Scity, Sstatus)
part     (Pname, P#, Pcolor)
spq      (S#, P#, Quantity)
```

Consider the query, "What suppliers supply part number one and in what quantity?" This can be expressed in Quel as follows:

```
RANGE OF s IS supplier
RANGE OF y IS spq
RETRIEVE (s.Sname, y.Quantity)
        WHERE     (s.S# = y.S#)
        AND  (y.P# = 1)
```

This query illustrates the three most common relational operators: *restriction*, *projection*, and *join*. The clause (y.P# = 1) is a restriction of the part relation to to only those tuples whose P# attribute contains the value 1. The clause (s.S# = y.S#) is the join condition. A join is a binary operation which creates a single new relation. Each tuple of the new relation consists of a tuple from each of the source relations such that the join condition is satisfied. Thus the join clause in the above query will, for each tuple in the supply relation and each tuple in the spq relation with the same S# field add to the result relation the concatenation of the two tuples. Finally, the clause RETRIEVE (s.Sname, y.Quantity) *projects* the Sname and Quantity domains out of the result of the join. That is, a projection selects only those domains out of a relation which are of further interest.

Query optimization is the process of transforming a query expressed in some non procedural language such as Quel into a sequence of local lower level database actions so as to minimize some measure of "cost". Distributed query processing is the retrieval of data from the various storage sites in the network. From the point of view of an interactive user, the cost is primarily the elapsed time required to process the query. From the point of view of the system as a whole it may be preferable to maximize throughput by minimizing the total resource usage. In single machine databases this may mean minimizing the number of page fetches. In geographically distributed databases with low bandwidth, high delay communication media, it often suffices to minimize the volume of data sent over the network. In local network databases, network traffic is no longer so simple a component of elapsed time. Whereas the communications links are several orders of magnitude slower than processing and disk transfer rates in long haul nets, in local nets, these rates are much closer. The opportunity exists to trade increased network traffic for increased parallel processing and consequently

improved response time. Furthermore, the cost of processing joins is worse than linear in the size of the relations while transmission costs are essentially linear. Thus for large amounts of data, we are given even more incentive to look for heuristics beyond simply minimizing network usage. In order to evaluate competing alternative strategies, an optimizer must be able to compute some heuristic function which approximates the true cost of running a query for the given environment.

The earliest work in distributed query processing was done by Eugene Wong for the SDD-1 system [Wong 77, Bernstein 81b]. Wong's algorithm used a hill climbing approach which made step by step improvements to an initial feasible solution. It used a general cost function and could minimize total time or response time. This method yields efficient but not optimal access plans because of its susceptibility to local maximums. The final version actually implemented in SDD-1 emphasized the use of semi-joins (see section 3.4.2), used zero processing costs, and considered only total time [Goodman 79]. Distributed Ingres [Stonebraker 77, Epstein 78] uses a query processing strategy known as *fragment and replicate*. This is a modification of the single site Ingres decomposition strategy [Wong 76]. This strategy tries to get a high degree of parallel operation by fragmenting one relation and replicating all the others and running the query in parallel at each site containing a fragment.

The solution space for distributed query processing strategies is defined by Chu and Hurley [Chu 79]. They show how to generate the set of equivalent query trees for a query by permutation of relational operators. Many of these trees are eliminated from future inspection by the application of theorems about optimal query trees. The trees are then transformed into graphs in which each node represents a set of operations performed at a single site. The arcs between nodes can be labeled with communications costs since no two adjacent nodes represent operations occurring at the same

site (otherwise they would be combined into one node). The nodes themselves can be labeled with processing costs. The graphs are scored based on the sum of their communications and processing costs, and the best one is picked. This yields an optimal strategy but is inherently exponential in the worst case.

For complex queries or small databases, the space of possible graphs and the number of variables affecting response time are so large that the cost of finding the optimal plan may dwarf the cost of running the query. Furthermore, determining the optimal strategy requires extensive statistics about the distribution of data in a relation. We must be able to predict accurately the number of tuples that are likely to result from any node in the graph. Thus we cannot hope to guarantee the choice of the optimal access path in a real-time environment. We can, however, employ heuristics which will generate reasonable access plans. Furthermore, complex queries are the exception rather than the rule. Most user queries tend to involve only a few relations and for these queries it may well be beneficial to find the optimal plan.

1.3 Purpose of the Thesis

The primary goal of this thesis is the implementation of a remote query processing mechanism for Ingres [Stonebraker 76]. This implementation is operational in the LOCUS distributed operating system [Popek 81] and allows a standard single site Ingres module to perform subqueries at remote sites. This remote query processing mechanism was implemented solely by this author over a five month period. However, this implementation depends heavily on the work of many other people. Most importantly, this work relies on the network transparent operating system environment developed by Dr. Gerald Popek, Dr. Bruce Walker and the LOCUS project at UCLA. This work is also based on the Ingres database system produced at The University of

California, Berkeley.

The first motivation for this implementation was the desire to create an environment for the investigation of query processing techniques in local network environments. To our knowledge, there are no fully operational distributed local network databases. Thus work on query processing has proceeded without an understanding of the local network environment. Most query processing research has assumed an environment in which communications are orders of magnitude slower than database operations. Consequently, algorithms have been able to solve the cost equations at the extreme where disk retrieval and local processing are free. The assumption that disk and processor costs are insignificant is certainly inappropriate for the local network environment. However, little was previously known about the cost characteristics of query processing in high speed networks. At one extreme, a local network may be so fast that communications costs may be considered to be free. At the other extreme, the time required to set up messages, process interrupts and retransmit collisions may be so great that message traffic is still the bottleneck. In between is a large middle ground in which communications, processing, and disk requests all contribute significantly to the total cost of evaluating a query and must all be considered. Measurements of the performance of our distributed database system, reported in chapter 4, determine the relative importance of communications, processing, and disk costs.

The second motivation for this implementation of a remote query processing mechanism was to address the issue of the difficulty of developing software for a distributed system. For a variety of reasons, building software in a distributed system is often more difficult than building similar software for a single machine. First, in most networks, a very different procedure is required to access data stored on a remote machine from that used for local data. A distributed data base must build a mechan-

ism to get remote files and to coordinate updates to replicated copies. Whereas in a single machine the database could use commands like open, close, read, and write, for remote data a more complex protocol such as ftp (file transfer protocol) is usually required. Second, software for distributed systems must deal with much more complex error conditions than corresponding applications on single sites. In single site systems, a machine crash results in a total restart of the entire system. In a distributed system, we are faced with partial system failures. That is, one or more of the sites cooperating on a task may fail leaving the others to clean up and continue. Furthermore, one of the primary motivations for installing a distributed system as opposed to a single large machine is the potential for reliability presented by the redundancy present in distributed systems. Unless the database is smart enough to continue executing using redundant resources, reliability will be reduced by the use of multiple machines.

A well designed distributed database should appear to users like the single site database they have grown used to. Thus it must handle all of the problems arising from the distribution of data: concurrency control, recovery from failures, multiple copies of data, distributed catalog management and distributed query processing. These problems make the design and implementation of distributed databases very much more difficult than analogous centralized systems.

A potential solution to this problem is to present the application, in this case the database, with a view of the world that is like a single machine. The complexities due to the existence of machine boundaries are pushed below the level visible to the database. This concept is known as network transparency.

The LOCUS operating system [Popek 81] begins with many of the same goals as a distributed database. The designers believe that just as a database user need not

concern himself with the dispersion of data, a computer user in general should not have to be aware of the existence of the network. A LOCUS network appears to the user like a single computer running Unix. In order to achieve network transparency, the system must provide facilities to handle the problems that arise from the distribution of data and processing power. Thus, like a distributed database, LOCUS handles concurrent access to files, distributed directory management, replication, transactions, partitioned operation and recovery from failures. In fact, LOCUS takes care of many of the problems that make building a distributed database difficult. This overlap between the mechanisms required by a database and an operating system occurs because the problems that the mechanisms address are inherent not to databases or operating systems, but to distributed systems in general.

Because LOCUS provides many of the mechanisms that distributed databases require, it would appear to be a good environment in which to build a distributed database. In fact, since LOCUS is application code compatible with Unix, any database that runs under Unix will become trivially distributed on a LOCUS network. Ingres, which was developed for Unix, runs on LOCUS without modification. It accesses remote data, has replicated relations and works in the face of partial failures.

That single site databases become distributed so easily in a LOCUS network is no small feat but they are distributed databases only in a limited sense. Only a fraction of the potential benefits of the network are harnessed. Because the data is not all local, the site running a query must bring all data across the network for local processing. The many other processors in the system are wasted. Not only could the use of remote processors speed the query by operating on portions of the job in parallel, but it could also reduce the amount of data transmitted over the network. Furthermore, distributed databases give very large numbers of users access to the data. Since all

queries on a given database must reference the same system catalogs and many of those queries modify the catalogs, their concurrency control may cause serious bottlenecks if not redesigned for a distributed system. Though most applications may be unaware of the existence of the network, databases are in some sense special users. They have both the requirement and the ability to make intelligent decisions based on the location of data. Thus the operating system provides the tools to control the location of a file or process.

The purposes of this thesis are twofold. First, the thesis provides some empirical results concerning the operation of a database in a high speed local area network. Using these results, conclusions are drawn about the performance of alternative query processing algorithms in such an environment. Second, this thesis tests the hypothesis that LOCUS is a good platform on which to build a distributed data management system. It proposes a new architecture and method of development for distributed databases and demonstrates its potential.

1.4 Thesis Plan

The outline of the rest of the thesis is as follows. Chapter 2 introduces the notation and distributed database model considered in this work. Chapter 3 presents the problem of query processing in distributed databases and reviews several algorithms for optimizing distributed access plans. Chapter 4 discusses the local area network as an environment for distributed databases. It reports the quantitative results of our measurements of a local area network database and derives values for the parameters of the cost model. Finally, it discusses the implication of these results for query processing algorithms.

Chapter 5 presents a case study of the construction of a local network distributed version of Ingres in LOCUS. Details are provided of the implementation and performance of the remote subquery processing mechanism for Ingres which allows many processors to cooperate in processing a query in parallel. This method of constructing a distributed data management system is compared to the Distributed Ingres effort at the University of California, Berkeley. Chapter 6 presents the conclusions to be drawn from this work and discusses the opportunities for further research. Finally, Appendix 1 gives a more detailed report of the methods and results of the measurements discussed in Chapter 4.

CHAPTER 2

Distributed Database Model

2.1 The Network Model

The distributed database is implemented on top of a computer network consisting of sites $S_1, S_2, ..., S_N$. Each site in the network is an autonomous computer. We will assume for purposes of this thesis that each cpu is identical though it presents no great problem so far as query processing is concerned to model heterogeneous hardware. Each of the N sites in the network is running a fully functional copy of the distributed database software which is capable of originating queries or servicing remote requests. The sites are linked by an interconnection network. We are considering both long haul networks such as the Arpanet and high bandwidth local nets. In general, the network may contain gateways between subnets of differing technologies and bandwidths.

We assume that the time (cost) to send data between any two sites is a function of the volume of data only and not of the identity of the sites. The volume of data is measured in pages where each message can send up to one page. We denote the cost of sending a page of data across the net by the constant C. The notation $C_K(x)$ is used in the discussion of the Berkeley Distributed Ingres proposal to denote the cost of sending x pages to K sites. Distributed Ingres considers the possibility of using broadcast or multicast messages to reduce the number of messages. We use this notation so

that it is clear that the time elapsed in sending data to many sites is the same as for one site if broadcast capabilities are available. While the underlying technologies in Ethernets and ring networks are broadcast media, the protocols required for reliable operation impose a site to site discipline on the network. Thus the discussion of Berkeley's proposal for broadcast nets is not extended to other methods. Finally, we assume that the network successfully delivers all packets correctly and in order.

2.2 The Database Model

The distributed database consists of a collection of relations $R_1, R_2, \ldots, R_m$. A relation may be stored locally (at the site originating the query), remotely (all tuples at one remote site), or horizontally fragmented [Rothnie 77] over some subset of the sites in the network.

2.2.1 Horizontal Fragments

An horizontal fragment is a subset of the tuples in a relation. We will call the jth fragment of relation R_i, R_{ij}. Distributed Ingres [Stonebraker 78] assumes that the union of all pieces of R_i contains no duplicate tuples and thus that fragments are disjoint. Thus when a tuple is inserted into a fragmented relation in Ingres, the update is performed in one fragment only. This restriction is unfortunate as it restricts a site's autonomy to choose which subset of tuples it will store locally. As discussed in section 3.3.3, the Ingres definition of fragments results from its particular query processing strategy. We know of no fundamental reason why fragments must be disjoint.

Fragments may be assigned to specific storage sites in several ways. The most common way in which this will happen is that the relation is created with a distribution criterion. A distribution criterion is a list of qualifications for each site storing a

fragment. Each tuple that is to be added to the relation is checked against the qualifications at each site and stored where it is appropriate.

The distribution criteria may be used by the query optimizer in selecting an access strategy. [Selinger 80] shows that a query on a fragmented relation may be transformed into a union of the results of that same query run on each of the fragments. Then the distribution criteria may then be used to eliminate some of the subqueries before they are run. For example, consider an employee relation fragmented such that S_1 stores all tuples for which <State = "New York">, S_2 stores all tuples for which <State = "California">, and S_3 stores tuples for which <State ≠ "New York"> AND <State ≠ "California">. In order to process a query that asks "retrieve all tuples with <State = "Maryland">", it is can be discovered a priori that only tuples on S_3 could satisfy the query. The query optimizer may check the distribution criteria and then export a subquery only to those sites whose distribution criteria allow them to store tuples satisfying the qualifications list.

Furthermore, the distribution criteria must depend only on attributes in that relation. Joins with other relations cannot be used to define fragments. This restriction causes no loss of generality at this level for two reasons. First, equivalent functionality can be provided to the user via a view mechanism implemented as a preprocessor on top of the query optimizer. Each fragment is actually a separate relation as far as the database is concerned but is mapped into a single relation by the view mechanism. Second, query optimizers are not able to make efficient use of multi-table distribution criteria to optimize joins anyway. It is not too difficult to check if a single relation query contradicts the distribution criteria at any site. However, if we allow tuple placement to be dependent on the contents of other relations, a query must be run in order to check for conflicts with the distribution criteria. This may be acceptable at

update time but it is not reasonable for the optimizer to run queries on other relations when doing retrieves.

Distribution criteria for fragments will not always exist. Tuples may be assigned to storage sites at random, so as to balance the number at each site, or on the basis of where they originated. If no criteria exist, a query processor will have to either assemble all of the fragments at one site to run a query or invoke a subquery at each of the possible storage sites.

2.2.2 Replicated Copies

An exception to the above requirement for disjoint fragments is made to allow the database to contain replicated copies of relations or fragments. The specific copy of a relation or fragment will be named in the model by the site number in the superscript. That is, the copy of the jth fragment of relation R_i that is stored at site 1 is called R^1_{ij}. It is important to note that R_{ij} is a logical data name whereas R^1_{ij} names a specific physical copy of the data. The superscript affects only cost or performance and has no effect on the logical result of a query. As will be discussed, most query processing strategies map the replicated database into a *materialization* before optimization begins. A materialization is a mapping of each logical relation name to a single physical instance of that relation. If only one copy of a relation or fragment exists or is being considered the superscript is omitted.

2.3 Database Cost Model

In our cost model of the distributed database, we must consider the three major resources used in retrieving distributed data. The cost is a weighted sum of the processor time used, the network bandwidth used, and the time required for the disk to ac-

cess the data.

$$TotalCost = DiskCost + ProcessorCost + CommunicationsCost$$

The *DiskCost* is incurred at each page fetch.

$$DiskCost = D * \#pagefetches$$

where D is the cost incurred in retrieving one page from the disk. The *ProcessingCost* is incurred with each tuple requested.

$$ProcessorCost = P * \#tuplerequests$$

where P is the cost per tuple processed. For example, joining two relations, say R_1 and R_2 with 10 and 100 tuples respectively using a nested loop involves 1010 tuplerequests.

As discussed above, the communications cost is proportional to the number of pages (one message per page) sent across the network.

$$CommunicationsCost = C * \#messages$$

C is the cost per message which includes the transfer time for the data, the line turnaround time and the instructions required to set up and receive a message.

The relative sizes of these cost constants characterize the database environment. In a database like SDD-1 which works over the Arpanet, the value of C is many times that of D whereas in a local net they may be quite close. In Chapter 4 and Appendix 1 methods and units for characterizing these weighting factors are discussed. Values are obtained for the constants P, C, and D for both a long haul network and a local network database.

It is not being suggested here that this cost function is the optimization function that should be minimized by the query processing algorithm. Rather, this function is a total resource usage function. It's usefulness is in illustrating how each of the three major resources contribute to the total cost of processing a query. For example if it turns out that the total cost of processing a given query by method A is 100 and by method B is 150, we do not conclude that A is superior to B. Method B may utilize more sites in parallel, increasing this total cost function while actually running faster. What is of interest is the fraction of the total cost due to each component. If in plan A for a given environment only a few seconds of network time are required, while the disk and processor time are high, it indicates that methods which would trade off more network traffic for more parallel processing should be preferred. On the other hand, if the 100 cost units is 95% due to communications overhead, this would indicate that algorithms which minimize network usage are appropriate.

As long as none of the three resources is nearing 100% utilization, this total cost function is a valid indicator of the percentage of cost due to each resource. However, if one resource is a bottleneck, this model must be extended to consider queueing effects. In the local network environment, the only candidate for such a model is the disk system. Developing a queueing model of query processing is a difficult undertaking in itself and will not be attempted here.

The next chapter examines proposed solutions to the problem of query processing in distributed databases. The solution space is defined and then several algorithms are presented. The algorithms chosen for review here are selected because they are actually implemented or close to implementation.

CHAPTER 3

Query Processing in Distributed Databases

3.1 Introduction

The basic problem in evaluating a relational query in a distributed database is that the information required may be on different machines. If the information required is all on one machine, then the problem reduces to that of determining an efficient access path in a single site database. Assuming the query does span multiple machines, the boundary between machines may occur between relations, between tuples within a relation (horizontal fragmentation) or between attributes of a relation (vertical fragmentation) as well as in combinations of these three. Let us ignore for the moment the problems of redundant data and fragmentation.

We are left with the general problem, that is, finding an access plan for multivariable queries that require joins on relations that span machine boundaries. The existence of the boundaries introduces three complexities not present in the single machine environment. The first is that we will inevitably have to move some information over the network, introducing a potential cost not present in single machine databases. The second is that we now have many redundant processors which can work in parallel in evaluating a query, introducing a potential to decrease response time which we did not have in single machine systems. The third complexity (which is being ig-

nored for the moment) is that redundant copies of data are also available, giving many more options for accessing the needed information. So, our problem is to use the options presented by redundancy to find in a reasonable amount of time, an efficient way to combine information that spans machine boundaries.

The problem of finding the optimal query processing strategy has been shown by [Hevner 79] to be NP-hard. The work of Chu and Hurley, reviewed in the next section has shown how to generate the entire solution space and how the time to search it can be reduced. However, most strategies use heuristics to avoid the exponential complexity of searching the entire solution space. These strategies do not produce true optimal solutions because they are susceptible to local minimums and are based on possibly erroneous estimations of result relation sizes. However, they produce solutions that are dramatically better than naive approaches [Epstein 80b]. Two of these heuristics are described below.

As we have said, if data is dispersed, information must eventually be sent across the network; we cannot avoid some communications costs. However, we can hope to reduce the volume required. The first technique is to perform all unary operations as soon as possible. The costs of performing unary operations are minute as they never involve intersite transmissions and the time required to process a Select is almost entirely the cost of retrieving the relation from disk. The benefits from this strategy arise from the reduction properties of unary operations. The input volume of data is always greater than or equal to the output volume so they serve to reduce the scope of the query. Just how much the size of the relations in the range is reduced by the unary operators depends on the distribution of the values in the attributes being restricted and various techniques have been proposed to estimate the cardinality of the result relations. [Epstein 80b] has shown through simulations that the plans resulting

from optimization using these estimation techniques are only slightly inferior to plans derived from perfect knowledge.

Secondly, it is often beneficial to execute all joins that can be executed with only local computation. If these relations must be joined, chances are that there is no cheaper way to accomplish the join than to perform it directly. This is of course, an heuristic and not a theorem as it is possible to conceive of examples where it might have been cheaper to first join one of the local relations with a remote one in such a way as to greatly reduce its cardinality. However, the benefits in the normal case far outweigh the increased costs in these worst case examples. The advantage of this heuristic is that it reduces the number of relations remaining in the range of the query from n to n-1 which reduces the number of possible join orders by a factor of n (in the absence of replication). By the above heuristics, the task of the distributed query optimizer is reduced to the problem of distributed join optimization.

3.2 Searching the Solution Space

There are two phases to the query optimization task as it is implemented by most algorithms. The first phase generates and compares possible join orders (called J-strategies) which may be viewed as programs for an abstract join machine [Reiner 82]. These programs are represented by query trees in which the leaf nodes are relations and internal nodes are either joins or temporary result relations. The second, or P-strategy phase generates programs for an abstract physical machine. These strategies are generally represented as computation graphs in which nodes are groups of operations and arcs represent data movement. This phase considers the many different join methods available for each join in the query trees. For example a typical abstract physical machine might implement scan, join, sort, move, and create index operators

[Rosenthal 82]. A cost model allows the optimizer to choose between the strategies in the P domain. A query optimization algorithm is characterized by the subsets of the P and J spaces it considers and the cost model it uses to judge alternatives.

Chu and Hurley [Chu 82] show how to generate the entire space of feasible query optimization strategies. They start with an initial feasible solution or *ifs*. A simple ifs is obtained by transmitting the entire database to a single site and running the query there as if it was a single site database. They describe algorithms to derive the entire space of J-strategies from the ifs and subsequently to obtain all possible P-strategy graphs.

The initial feasible solution is represented by a query tree in which each leaf is a storage node for a relation, each internal node is a relational operation, and the root is the result. Arcs linking the nodes indicate data flow from the leafs towards the root of the tree. Using the commutative, associative, and distributive properties of the relational operators the set of feasible query trees that produce the correct result are generated.

If the goal is simply to insure that the optimal solution is generated, their algorithm prunes the set of feasible query trees using *theorem 1* from [Chu 82].

> *Theorem 1*: If the unit communication cost between any two sites in the distributed database is the same, and the processing cost of a given operation is the same for all computers and is proportional to the volume of data, then placing each unary operation at the lowest possible position in a query tree is a necessary condition to obtain the optimal query processing policy.

This theorem is a formal statement of the common wisdom that it is desirable to perform unary operations first. The more expensive binary operations (joins) are postponed in the hope that the unary operations will reduce the volume of data and thus make the joins cheaper.

In order for this theorem to be used in a real database, we must add a further stipulation. Suppose our initial feasible solution calls for a join of R_1 and R_2 on attributes d_{11} and d_{22} where d_{11} is a primary key for R_1. Suppose further that each relation contains 10 tuples and R_1 has an unclustered index on d_{11}. In order to perform this join, we would scan R_2 sequentially and for each tuple, we would get the corresponding R_1 tuple using the index. This will require we fetch each tuple once for a total of 20 tuple requests. Now consider the case if the query contains a clause restricting R_1 on some other domain d_{12}. If this restriction is performed before the join as the above theorem dictates, we would lose the index on the join field. Without the index, the join requires 110 tuple requests. Thus we will state our version of the theorem.

> *Theorem 1a:* If the unit communication cost between any two sites in the distributed database is the same, and the processing cost of a given operation is the same for all computers and is proportional to the volume of data, then for each unary operation on relations which do not have an index on any domain named elsewhere in the query tree, placing each unary operation at the lowest possible position in a query tree is a necessary condition to obtain the optimal query processing policy.

Given the set of feasible query trees in the J-space, Chu and Hurley's method then transforms them into the complete set of query processing graphs. Each node of the graphs represent a set of operations executed at a single site. Each arc represents a

transmission of data between sites. The graphs are translated into query processing policies by mapping each node to an execution site. Leaf nodes are mapped to their file storage sites and three theorems generate possible mappings of internal nodes to sites. Again, application of another theorem can eliminate graphs which could not possibly represent an optimum plan.

This reduced solution space may then be searched for the optimal plan. Each graph is scored based on the cost function and the lowest cost graph selected. This method guarantees an optimal plan because the entire solution space is considered. While it is still inherently exponential, the algorithm performs in only a fraction of the time of a complete exhaustive search. Furthermore, it can be easily extended to cover semijoin strategies by generating additional graphs at the final step in which each join is replaced by the corresponding sequence of a semijoin followed by a join. In a two phase optimization strategy such as this, the abstract physical machine can be extended with additional join methods such as semijoins without affecting the first phase.

3.3 The Distributed Ingres Proposal

Ingres (Interactive Graphics and Retrieval System [Stonebraker 76]) is a database management system implementing the relational model of data [Codd 70]. Current Vax Ingres consists of two user processes on top of a Unix or LOCUS operating system. The first process, the "monitor", is either an interactive interface to the user's terminal or an applications program with embedded EQUEL statements. It formulates a character string representation of the query and passes it down a pipe to the second process. The second or "main" Ingres process interprets the query and prints the results on the user's terminal.

Berkeley Distributed Ingres consists of a monitor and a collection of main Ingres processes. The monitor is connected by a pipe to one of the main processes which is the master. The master receives the character string representation of the query and runs the query decomposition module. This module has the ability to cause subqueries to be executed on other sites and to move relations or fragments between sites via the Unix version 4.2 interprocess communications system. For a more complete discussion of the system architecture see [Stonebraker 76].

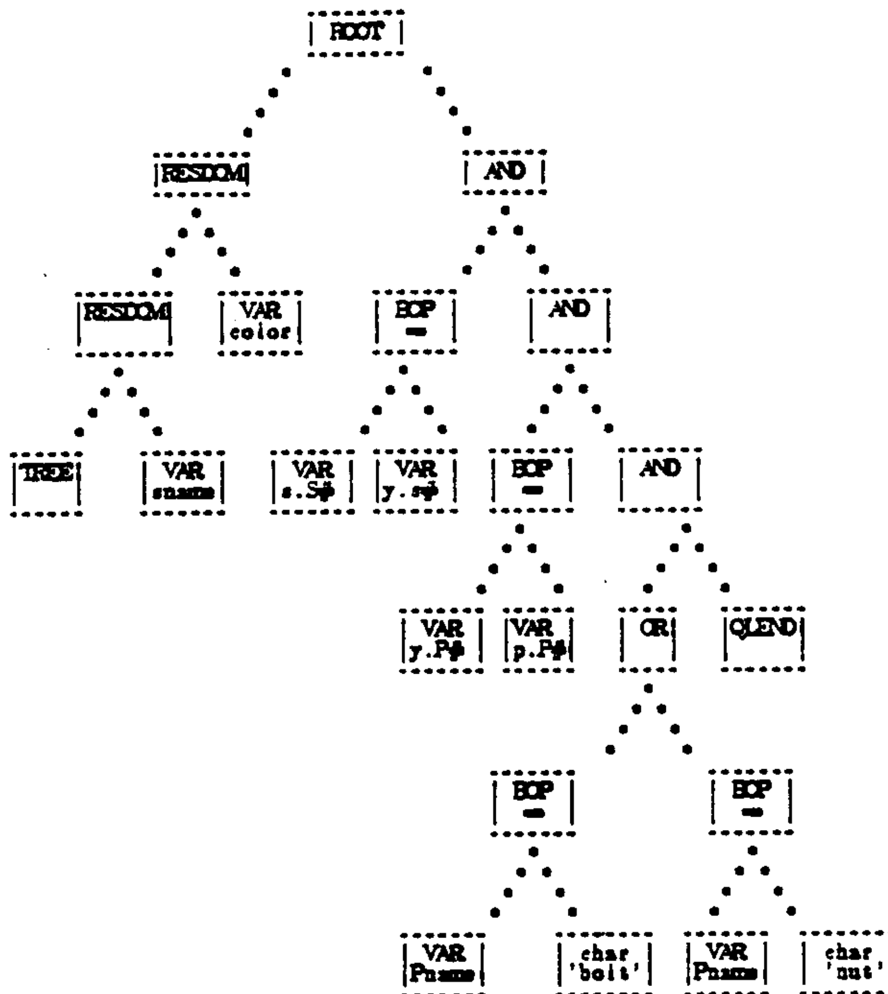
3.3.1 Query Processing in Single Site Ingres

Epstein's Distributed Ingres query processing proposal [Epstein 78] is an extension of the single site algorithm. Thus it is essential to understand the standard Ingres approach before proceeding to the distributed version.

Ingres builds an internal binary tree representation of the query. The left branch of the root node specifies the list of result domains (RESDOMS), also called the target list. The right branch of a RESDOM is a VAR node while the left branch is either the next RESDOM or an end of list marker (a TREE node). A VAR node names the attribute to participate in the result. It also contains pointers to a copy of the relation relation tuple and the attribute relation tuple for the attribute to be used in processing the query.

The right side of the tree is the list of qualifications clauses in conjunctive normal form. It is represented by a list of AND nodes. The left branch of each AND is a BOP node which specifies a single qualification while the right branch either heads the remainder of the list (another AND node) or marks the end of the list (a QLEND node). A BOP has an operator such as '=' or '<'. Its left branch is a VAR node while its right branch is either another VAR node or a value with a type specification.

The interpretation of the clause is that the statement "left child operator right child" must be true for any tuple surviving the qualification. If both children are VARs from different relations the BOP node specifies a join condition. If one child is a VAR and the other is a constant, the qualification is a simple restriction such as "salary > 20000".




```
RANGE OF s is supplier
RANGE OF p is part
RANGE OF y is spq
RETRIEVE (s.Sname, p.Pcolor)
        WHERE (s.S# = y.S#)
        AND   (y.P# = p.P#)
        AND   ((p.Pname = 'bolt')
        OR    (p.Pname = 'nut'))
```

Figure 1: Ingres Query Tree.

Ingres uses trees to represent queries and subqueries internally. Future examples of subqueries will use QUEL but it should be understood that it is really parse

trees that are being manipulated internally.

The above tree specifies the query but it says nothing about how it is to be processed. One method that would always work is to form the cartesian product of all the relations in the range statement. Then test each tuple against each of the qualifications and remove ones that do not satisfy all qualifications. Finally project out those fields specified in the target list and eliminate duplicate tuples. However, the cost of this simple method can be enormous. Consider the query in the previous example on the database of a small manufacturing company. They might have 100 different suppliers supplying 1000 different parts. With only a 10% overlap in what suppliers supply there would be 10,000 spq tuples. Thus the cartesian product of the range contains 1,000,000,000 tuples. At, say 100 bytes per tuple, processing this query would require 100 gigabytes of temporary storage to say nothing of the time required to scan such a relation.

In an interactive system such as Ingres, query "optimization" is not possible. There are simply too many possible ways of attacking a query and the real cost is too complex. Instead, the database searches for a suboptimal algorithm. Using several "rules of thumb" that work in all cases, Ingres breaks down the query into pieces that are always simpler than the original query. When the pieces are broken down into sufficiently simple subqueries, they are individually optimized. Thus it uses a "greedy" algorithm in which each piece is optimized without looking at the global structure of the query. While it is capable of generating any arbitrary query tree, the greedy lookahead scheme causes it to examine only a very small number of alternatives [Reiner 82].

Ingres uses a strategy called decomposition to break down a query. Decomposition has two main objectives: Firstly, that there be no cartesian product formed; And

secondly, that there be no geometric growth. That is, that the number of tuples scanned be kept to a minimum [Wong 76]. Decomposition uses two main tools:

1. Given a query with a range of n relations, it can be replaced by a set of queries on n-1 relations. This is accomplished by substituting, tuple by tuple, for the relation removed. This process is called *tuple substitution.*

2. A query Q can be broken into a sequence of of two queries Q1 and Q2 where Q1 and Q2 have only one relation in common. This is called *reduction.*

   Consider the query, "Retrieve all suppliers who supply part number 1".

```
RANGE OF s is supplier
RANGE OF y is spq
RETRIEVE (s.Sname) WHERE (s.S# = y.S#)
                   AND   (y.P# = 1)
```

Using method 2 above, this query can be broken into two pieces that overlap on only one variable.

```
RANGE OF y is spq
RETRIEVE (y.S#) INTO temp
                WHERE (y.P# = 1)

RANGE OF s is spq
RANGE OF t is temp
RETRIEVE (s.Sname) WHERE (s.S# = t.S#)
```

The first has a range of only one relation and can be submitted directly to Ingres' one variable query processor (OVQP). The second may be attacked by tuple substitution. Suppose the spq relation contained the following tuples:

spq

S#	P#	Q
1	1	10
2	1	100
3	1	14
2	3	1

The result of the first query would be:

temp

S#
1
2
3

Tuple substitution would decompose the remaining query into three pieces in which the variable t.S# has been replaced by the three values in temp.

RETRIEVE (s.Sname) WHERE (s.S# = 1)

RETRIEVE (s.Sname) WHERE (s.S# = 2)

RETRIEVE (s.Sname) WHERE (s.S# = 3)

The Ingres query decomposition algorithm is a combination of the two techniques of reduction and substitution. The Reduction module divides the query into irreducible components. Subquery Sequencing turns the result of Reduction into a sequence of calls to Tuple Substitution. Tuple Substitution calls Variable Selection to select which relation to substitute for. The entire decomposition process is repeated recursively for each subquery generated until only one variable subqueries are left. These one variable queries are executed by the One Variable Query Processor.

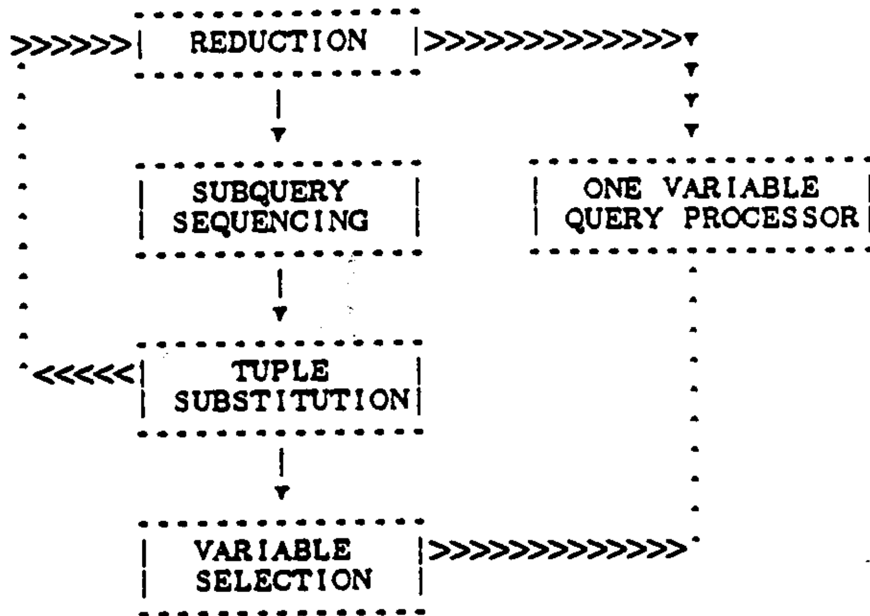


Figure 2: Ingres Query Decomposition Calling Sequence.

It is important to note that tuple substitution and detachment of overlapping subqueries represent no optimization in and of themselves. Tuple substitution is tantamount to forming a cartesian product in the number of tuples scanned if not in storage requirements. The optimization is embodied in the Variable Selection module. Firstly, it performs all one variable subqueries before doing any tuple substitution. Thus it reduces the cardinality of the join relations. Consider the query "Retrieve the names of all suppliers supplying part number 1." Assume the cardinalities of supplier and spq are 100 and 10,000 respectively and that there ten suppliers who supply part 1. If the select out of spq is done before the join of spq and supplier, 10,000 + 10(100) = 11,000 tuples are examined. If, however, the join is done first, 100(10,000) = 1,000,000 tuples must be examined; a difference of two orders of magnitude. Secondly, the Variable Selection module decides which relation to substitute for. This is equivalent to deciding which variable to make the inner and which the outer in a nest-

ed loop. Suppose we must join the 100 supplier tuples with the 10,000 spq tuples. Further, assume that there are 10 tuples per disk block and that we can keep only 10 blocks in core at a time. The entire supplier relation will fit in core while spq will not. If spq is the variable substituted for, it will require 1000 + 10 = 1010 block fetches. If, however, we substitute for supplier, it will require 10 + 100(1000) = 100,010 block fetches to do the join. The intelligence is embodied in the heuristics of the Variable Selection module. These heuristics are detailed in [Wong 76].

3.3.2 Query Processing in Berkeley Distributed Ingres

[Stonebraker 76] presents a proposed set of extensions and modifications to Ingres to allow it to manage data distributed over multiple machines. We will examine only those issues related to query processing in such a system. The following sections are a review of the query processing strategy proposed in [Epstein 78] and [Epstein 80b]. The algorithms and equations reported here are the work of Epstein, Stonebraker and Wong and their implementation is underway at the University of California, Berkeley.

What the user interface to fragment definition and placement should look like is a very open question. Here we will use the additions to QUEL suggested in [Stonebraker 76] in which the location of data can be explicitly visible to the user.

Suppose we wish to define the supplier relation such that S1 stores all suppliers located in Los Angeles, S2 stores all suppliers in San Diego, and S3 stores the remaining suppliers. This is expressed in extended QUEL by:

```
RANGE OF s IS supplier
DIST_CREATE supplier (Sname = c16, S# = I4, Scity = c12)
      (s.Scity = 'Los Angeles' = S1,
```

s.Scity = 'San Diego' = S2,
s.Scity ≠ 'Los Angeles'
AND
s.Scity ≠ 'San Diego' = S3)

The above example will create a distributed supplier relation. The system will automatically insure that future updates are accomplished so as to meet the distribution criteria. Furthermore, the system will use the distribution criteria to optimize the query. To accomplish this, before forking a subquery at any site, it checks if any of the qualifications in the query contradict any of the distribution criteria. If the intersection of the set of possible tuples satisfying the query with the set of tuples that could exist at a particular site is obviously empty, the subquery is not run at that site. To handle the general case requires a theorem prover. However, given that distribution criteria are restricted to single variable statements, a simple mechanism will suffice for most cases.

3.3.2.1 The Distributed Ingres Algorithm

The basis of the distributed QUEL processing algorithm proposed in [Epstein 78] is the same as the decomposition procedure used in single site Ingres. It breaks the query down into simple pieces and optimizes them individually. The "master" Ingres site (the site from which the query was initiated) orchestrates the processing of the query. It uses the parse tree (Figure 1), the location of each fragment, its cardinality and distribution criteria, and the network type (point to point or broadcast). The master site in Berkeley's implementation, has at its disposal the ability to:

1. invoke a slave Ingres at site i to run query Q.

2. move the fragment R_{ij} to specified sites.

The first step of the algorithm is to perform all one variable subqueries. The two one variable (unary) operations considered are select and project. The number of tuples resulting from a select is always less than or equal to the number of tuples in the operand relation. Similarly the width of tuples resulting from a project is always less than or equal to the width of the input tuples. So for any unary operation, the output volume of data is always less than or equal to the input volume of data. Doing unary operations first reduces the cost of future binary operations by reducing the number of tuples in the input relations. Unary operations are quite cheap as compared to binary operations because they do not involve any network communications. There are examples in which postponing the unary operations is superior but the penalty in such cases is very small compared to the gain in the usual case.

Consider a database with a supplier relation fragmented as defined above and the query:

```
RANGE OF s is supplier
RANGE OF y is spq
RANGE OF p is part
RETRIEVE (p.Pname) WHERE (y.S# = s.S#)
                AND  (s.Sstatus > 10)
                AND  (y.P# = p.P#)
```

The restriction on the supplier relation is a one variable subquery which may be separated and run immediately.

```
RANGE OF s IS supplier
RETRIEVE INTO temp (s.S#)
      WHERE (s.Sstatus > 10)
```

The master Ingres site first checks that the qualification (s.Sstatus > 10) does not contradict the distribution criteria for any of the sites storing supplier. Since in this case

there is no conflict, it creates the temp relation distributed among s1, s2, and s3 with no distribution criteria. The master Ingres then invokes a slave process at each of the three sites storing fragments of supplier and instructs them to run the above subquery. After running the subquery, the remaining query is:

```
RANGE OF t IS temp
RANGE OF y IS spq
RETRIEVE (y.Pname) WHERE (t.S# = y.S#)
          AND  (y.P# = p.P#)
```

Since the query is expressed in conjunctive normal form, if any subquery was not satisfied by any tuples, then the whole query is false and we can quit. In our example there was only one one variable subquery and we will assume each of the three sites yielded some tuples.

The next step is to apply the reduction algorithm presented in the single site discussion [Wong 76] to the remaining query to produce a sequence of queries that overlap on at most one variable. Our example query is broken into:

```
RANGE OF t IS temp
RANGE OF y IS spq
RETRIEVE INTO temp1 (y.P#)
      WHERE (t.S# = y.S#)
```

followed by

```
RANGE OF t IS temp1
RANGE OF p IS part
RETRIEVE (p.Pname) WHERE (p.P# = t.P#)
```

The algorithm now selects the cheapest remaining piece of the query to process and decides whether to evaluate it directly or to subdivide it further. For example, if

the piece selected is a join on three relations it could move all relations to one site and perform the join there. Alternatively, it could join two of the three to produce a temporary relation and then join that to the third. Berkeley Ingres considers all possible ways of subdividing the query and compares the estimated cost of each to the estimated cost of executing the query directly. This exhaustive search is made possible by the assumption that the number of relations remaining in the range of an irreducible subquery is quite small.

Then Epstein's algorithm must choose which sites will process the piece selected. If the query can run at a collection of sites without moving any data, the master simply multicasts the query to these sites and lets it run. In our example, we need to join temp and spq where temp is fragmented between three sites and spq is at one site. Thus some data must be moved before the processing can proceed.

There are at least three options for our example. Since temp is distributed between three sites and spq is at one, we could multicast spq to the sites storing temp. The subquery joining temp and spq could then run in parallel at the three sites producing a distributed temp1 relation. Alternatively, we could choose to assemble the pieces of temp at the site storing spq. This might be the choice of preference if the spq relation is much larger than temp. Finally, we might want to equalize the load between the three (or more) machines by balancing the number of temp tuples in each of the fragments. Since the time to process the subquery is equal to the maximum of the time required at the individual sites, the advantage of load balancing is obvious.

In the centralized Ingres case choosing how to process the "next piece" amounted to choosing R_n the variable to substitute for given a query on relations R_1, R_2, ..., R_n. However, doing tuple substitution in the distributed case is enormously expensive

as it involves a transmission for each of the tuples in the relation substituted for. Sending small amounts of data across the net is in general inefficient because there is an initial setup cost for net messages which is independent of the volume of data. Rather than using tuple substitution, Ingres uses fragment substitution. The number, K, of fragments into which R_y is divided determines the degree of parallelism and the number of transmissions required to assemble the remaining n-1 relations at the processing sites. Furthermore, fragment substitution allows the optimizer to take advantage of the fact that R_y may already be fragmented so that no further transmissions may be required to distributed the substituted variable. The optimization problem is to determine K sites to be processing sites and R_y, the relation to be left fragmented.

Some criteria must be used for comparing the various strategies. Epstein's cost functions, allow Ingres to minimize either network communications or processing time for either point to point or broadcast networks.

3.3.2.2 Processing Cost Function

Epstein defines P(Q) to be the time required to process the query Q if K = 1; that is, if it is done on a single site. If a value of K greater than 1 is chosen, then some processing may be done in parallel. The time required for the whole query is equal to the maximum of the time required at each of the sites which is approximately proportional to the number of tuples from the fragment of R_y at that site, assuming the fragments are disjoint.

$$\max_j P(Q_j) = \max_j \frac{size(R_{yj})}{size(R_y)} P(Q)$$

It is clear that maximum parallelism and thus minimum processing time is achieved when the numbers of tuples from R_y at each site are equal. Balancing the number of

tuples in each fragment improves the processing time from

$$\frac{size(R_{rj})}{size(R_r)} \quad to \quad \frac{1}{K}P(Q)$$

where site j was the site with the largest fragment of R_r. The system must move tuples from R_r tuples so that each site has the same number. The additional amount of data that must be transmitted to equalize the fragments is given by

$$\sum_{j=1}^{N} pos\left[size(R_{rj}) - \frac{1}{K}size(R_r)\right]$$

where the function pos(x) equals x for x > 0 and 0 otherwise.

3.3.2.3 The Communications Cost Function

In the Berkeley Distributed Ingres work, the cost of sending x pages of data to K sites is denoted $C_K(x)$. This function differs for broadcast networks and point-to-point networks. In broadcast networks, the cost of sending x pages to K sites is the same as the cost of sending x pages to any number of sites whereas in point-to-point systems, the cost of sending x pages to K sites is K times the cost of sending x pages to one site. Assuming the relation R_r is already fragmented at K of the N sites in the network, then the communications cost is given by:

$$C = \sum_{j=1}^{K} C_{K-1}\left[\sum_{i \neq r} size(R_{ij})\right] + \sum_{j=K+1}^{N} C_K\left[\sum_{i \neq r} size(R_{ij})\right] + \sum_{j=K+1}^{N} C_1\left[size(R_{rj})\right]$$

Using these functions, the algorithm can compare network communication costs, processing times and the costs and benefits of equalizing the fragments of R_r.

3.3.2.4 Minimizing Communications Costs

The communications cost function is particularly simple for broadcast networks. The cost is minimized when K equals 1 or K equals the number of sites which store fragments of R_p. [Epstein 78] gives the following two rules for minimizing communications costs in broadcast networks:

1. If there exists a site for which the total of all relevant data stored at that site exceeds the size of the largest relation choose K = 1 and use that site.

2. Otherwise choose R_p to be the largest relation and choose K to be the number of sites which store fragments of R_p.

Minimizing communications in the point to point case is somewhat more complicated. The rules are:

1. Arrange the sites in decreasing order of the volume of data at that site. Thus site 1 contains the most data.

2. Let R_p be the largest relation.

3. Choose K = 1 and the processing site to be site 1 if

$$\sum_{i \neq p} \left[size(R_i) - size(R_{i1}) \right] > size(R_{p1})$$

   This says process at the site containing the largest volume of data if the number of tuples it must receive is greater than the number of tuples of R_p that it has.

4. Choose K to be the largest j such that

$$\sum_{i \neq p} \left[size(R_i) - size(R_{ij}) \right] \leq size(R_{pj})$$

This says a site should be chosen to be a processing site if and only if the volume of data that it must receive if it is a processing site is less then the additional data that it would have to transmit as a nonprocessing site. Sites 1 through j are processing sites.

3.3.2.5 Minimizing Processing Costs

In order to minimize the processing costs we must achieve maximum parallelism. This is achieved when all N sites in the network are processing sites and the tuples of relation R_p are equally distributed among all processing sites. Therefore, the algorithm chooses K equal to N. Furthermore, the formula given above for the processing time at each site in terms of P(Q) is independent of which relation is chosen as R_p. Thus Ingres can choose to leave fragmented the relation which minimizes the transmissions costs for N sites.

3.3.2.6 Running the Query

The last step in processing a query is to run the subquery at the selected sites. If the result is empty, the query processor can quit. Otherwise, it adjusts the ranges of variables remaining in the query to reflect the new temporary relations and goes back to the "select next piece" step.

1. Do all one variable subqueries.

2. Apply reduction algorithm.

3. For each remaining subquery do:

    4. Decide whether to subdivide further.

    5. Choose processing sites and relations to move.

    6. Move relations.

    7. Run the local queries.

3.3.3 Discussion of the Ingres Algorithm

The Distributed Ingres approach is known as the "fragment and replicate" strategy. It is derived from the centralized technique of decomposition [Wong 76] and its basic strategy is to fragment one relation and to replicate the others. All of the sites with fragments of the chosen relation can work in parallel to execute a join. This approach does not perform well in site-to-site networks or in unfragmented databases [Sacco 81]. In order to do a join, we must first distribute fragments of one relation about the network. This is expensive if the relation was not already fragmented. Then the remaining n-1 relations have to be sent to all of the K processing sites which, in the absence of a broadcast capability involves transmitting each relation K times. While technologies such as the Ethernet or the Token Ring are inherently broadcast technologies at their lowest levels, the requirements for reliability necessitate the imposition of protocols which effectively create point to point semantics.

The Distributed Ingres algorithm provides solutions to the problem of selecting K and R_p at the two extremes, minimizing processing costs and minimizing communications costs. Minimizing processing cost implies assuming that network usage is free

while minimizing communications costs implies that local processing is free. Assuming that one of the components of cost is zero greatly reduces the complexity of the optimization function and its solution. The quantity that we would really like to minimize is some linear combination of response time and total time, each of which are non trivial functions of processing time and network traffic. Minimizing response time is the goal from a individual user's point of view while minimizing total time maximizes the throughput of the system as a whole (in the absence of queueing delays). In slow networks like the ARPANET where sustained data rates are of the order of 10 kilobits/sec, the communications costs are a good approximation of real costs and the assumption of zero processing costs is reasonable. It is not satisfactory to minimize communications or processing costs alone in a local network database. Our results, reported in the following section, indicate that the most significant source of delay in query processing is neither processing or network communications but rather the retrieval of data from the disk. With the goal of reducing the cost of evaluating an access plan, Distributed Ingres has proposed a cost model that is insufficient to model the true nature of modern local area networks.

Distributed Ingres also does not attack the problem of fragmented relations in its full generality. It does allow the user to define distribution criteria which deterministically specify the site at which any given tuple will be stored, based either on the values of attributes within the tuple or on its creation site. The fragments of a relation must be disjoint in the Ingres distributed data model. Thus each tuple occurs only once and the union of the sets of tuples from all sites storing a relation form the set (without duplicate tuples) of all tuples in the relation. This restriction seems to be an unfortunate breach of local autonomy as it means that users cannot independently decide which tuples each would like to store at their sites. In an ideal system, one site

with a large amount of disk space and many users should be able to store all tuples while another with less space but the requirement for fast access of frequently needed data should be able to store some subset of interest to it. Stated differently, user A's desire to store tuples for suppliers located in Los Angeles on his workstation should not preclude user B from storing tuples for suppliers in Los Angeles or Denver. Storing relations fragmented in this way is not difficult. Each fragment is simply a different relation from the point of view of the lowest levels of the database. However, it greatly complicates the design of middle levels of the database which must make the many relation fragments transparently appear as one relation to the user. The most difficult problems occur in the directory design and the data definition interface.

It is important to note that the issues of fragmentation and replication are inextricably linked. Full relation replication is a special and particularly interesting case of fragmentation in which all tuples occur at each storage site. Alternatively, fragmentation can be viewed as tuple level replication. Replication has the potential to increase system performance and reliability. Distributed Ingres does not address this issue either, and understandably so as it is a very difficult problem. Consider the task of finding the optimum join order for four relations. The number of possible join orders is four factorial or 24 which is a reasonably small space in which to do an exhaustive search. Now consider the situation if each of the relations is replicated at three sites. The number of ways these can be joined is now given by

$$\frac{12!}{4(3!)} = 73,920$$

which is far too large to search exhaustively. The approach taken by most query optimizers is to form a "materialization" of the database independently of query optimization [Bernstein 81b], [Sacco 81]. Materializations of a database consist of exactly one

copy of each relation in the range of the query and are constructed using heuristics. These heuristics are poorly understood and are generally independent of the query characteristics and thus do not guarantee optimal schedules.

The site at which the result is needed is not considered by the Distributed Ingres proposal. In many cases, there is a final transmission cost to move the result to the destination site which is not considered in the cost calculation. It should be a relatively minor modification to the algorithm to include the final transmission cost in the evaluation of the access scheme.

Epstein's Distributed Ingres proposal does not consider the potential savings from the use of semi-joins which are a valuable tool in distributed query processing. The idea behind a semi-join is to send over the network only that data which will eventually participate in the solution. In order to perform a distributed equi-join on attribute A between R_1 at site 1 and R_2 at site 2 we could:

1. Project attribute A from R_1 and send the column to site 2.

2. Join R_2 with the column of R_1 and put the result in R_{temp}.

3. Send R_{temp} to site 1.

4. Join R_{temp} and R_1.

The above schedule, first does a semi-join of R_2 and R_1 to reduce R_2 to only those tuples which will actually join with R_1. It then performs the join between the R_1 and the surviving tuples of R_2. Semi-joins alone are not sufficient to evaluate arbitrary relational queries. When data resides in two relations that are at different sites, we must eventually transmit one of those relations and perform a join. The role of the

semi-join is to limit the communications cost required to perform the join as much as possible.

In [Bernstein 81] and [Goodman 82] the reduction properties of semi-joins have been investigated. A sequence of semi-joins is called a *semi-join program*. Any semi-join program that restricts the database to only those tuples that participate in the result (with respect to a given query q) is said to be a *full reducer*.

We define a query graph for a query q as follows. There is a node for each relation in the range of q. There is an arc between each pair of nodes i and j for which there is a clause in the qualifications list of q that references both R_i and R_j. The set Q of all queries is partitioned into two classes. A query is called a *tree query* (TQ) if its query graph is a tree. Otherwise its graph contains a cycle and it is called a *cyclic query* (CQ).

These are important distinctions because for all q which are elements of TQ there exist semi-join programs which are full reducers. For cyclic queries, not only does no full reducer exist for all database states, but no sequence of semi-joins that is even partially beneficial may exist. [Bernstein 81] gives a linear time algorithm for determining if a query is a member of TQ or CQ.

Semi-joins can result in a large savings when:

1. communications costs are high.
2. the join field is very small compared to the length of the tuple.
3. a small percentage of the tuples participate in the join.
4. the relations are large enough that in-transit time dominates startup costs.

[Sacco 81] suggests that semi-join strategies are extremely beneficial for queries in which a small number of semi-joins will fully reduce the query. The performance of semi-joins for cyclic queries and for queries that are not fully reduced by a few semi-joins is generally very poor. While semi-joins appear to be essential for the performance of systems that must contend with slow networks, we would not expect semi-joins to be very useful in local network databases.

3.4 Query Processing in SDD-1

SDD-1 is a Distributed Relational Database developed by the Computer Corporation of America [Rothnie 80]. It manages data geographically distributed among sites connected by the Arpanet. The database provides for redundant storage but the query mechanism considers only a materialization of the database. As network communications are considered to be the bottleneck in the system, the goal of the query processing mechanism is to minimize network communications.

3.4.1 The Hill Climbing Approach

The original SDD-1 algorithm [Wong 77] was one of the first distributed query processing algorithms published. The algorithm selects an Initial Feasible Solution (IFS) and then tries to make stepwise improvements. To construct the IFS the strategy is as follows:

1. Perform all local operations (selects and projects) that can be done without moving any data.

2. Make the minimum number of relation moves which will assemble all data at one site.

The idea behind this approach is that distributed query processing has to contend with two types of boundaries: between relations and between sites. Traversing machine boundaries (moves) incurs transmission costs and traversing relation boundaries (joins) incurs processing costs. Optimizing moves is the primary concern in the SDD-1 environment and thus Wong's strategy isolates the two problems. This is done by considering two different views of the database (denoted V_1 and V_2).

V_1: The user's view consists of the user data model plus some or all of the distribution information.

V_2: The Distribution view consists of one relation per site each of which is the cartesian product of the relations stored at that site.

Step 2 in the formation of the IFS above finds a transformation of the query on V_2 into a query on a single variable (site). Next the algorithm attempts to optimize this transformation.

Wong denotes by M_0 the sequence of moves compiled in step 2. The next step is to attempt to replace M_0 by two sets of moves, M_1 and M_2 that when executed sequentially with some local processing between them have the same result as M_0. The $\{M_1,M_2\}$ pair that decreases total cost the most is selected. The algorithm is applied recursively to each of the pieces until no further beneficial modifications can be found.

Like Ingres, this is a greedy algorithm and thus cannot possibly guarantee an optimal solution. Avenues which have high initial costs but which lead to greater overall savings at later stages may be discarded in favor of more immediate gains.

3.4.2 The Final Version

[Goodman 79] reports the algorithm as it was finally implemented in SDD-1. The final version of the SDD-1 algorithm differs from the first version in three important ways [Sacco 81]:

1. It recognizes the importance of semi-joins in distributed query processing.

2. It uses a simpler cost model (which is appropriate for the Arpanet) in which processing costs and message setup costs are zero and the communications costs are proportional to the volume of data.

3. Total time is minimized. (The first version could minimize total or elapsed time.)

The basic plan is in three steps and differs little from the original version. First, a program P of relational operations executed at the various sites to delimit a subset of the database. This phase determines an *envelope* which contains a superset of those tuples actually required to process the query. Next, this envelope is transmitted to a single site. Finally, the remaining query is executed at the chosen site. The optimization problem is primarily in the reduction phase and this is where this algorithm differs from the previous one.

A *reduction* of a database D with respect to a query Q is any database D' such that $Q(D) \leq D' \leq D$. A *reducer* for Q is a program P of relational operations such that for all databases D, P(D) is a reduction of D with respect to Q. The benefit of a reducer is the amount of data it removes from the database. Its cost is the amount of intersite transfer of data that the reducer requires. In this cost model, restricts and projects entail no cost as they require no data to be moved. Since they have no cost

and have positive benefits, our reduction stage applies all of the restrictions and projections allowed by the query. Any further reduction requires some transmission of data. As we have seen before, the semi-join is the ideal operator for such an environment. It can limit the scope of the database without transmitting any complete relations. Furthermore, semi-joins always result in a non-negative benefit whereas joins can actually increase the size of the envelope. The complete algorithm is as follows:

1. P = the sequence of all local operations permitted by the envelope, E.
2. Estimate the cost and benefit of all non local semijoins permitted by E.
3. Do while some non local semijoin has benefit > cost.
    4. Let sj be the most profitable semijoin.
    5. Append sj to P.
    6. Estimate the reduction effect of sj and update the cost and benefits accordingly.
7. End loop.
8. For each site S let *size*(*S*) equal the sum of the sizes of all relations stored at S and referenced by Q.
9. Select S_a to be the site with maximum *size*.
10. Append to P commands to move data from all other sites to S_a.
11. End.

There are at least two simple enhancements to this method [Bernstein 81b]. The first enhancement is to reorder the semijoins within the program in such a way so as to reduce the cost of doing some without increasing the cost of any others. This is accomplished by delaying some expensive semijoins until the relations involved have been reduced by other less expensive semijoins. This compensates to some extent for

the lack of backup in the search algorithm.

Bernstein's second enhancement takes care of mistakes that can be made as a result of having chosen the sequence of reductions before picking the site at which the processing will be done. We can prune from P those semijoins that are rendered unprofitable by the choice of S_r.

It is important to note that the SDD-1 strategy compiles queries whereas Ingres interprets them. A compiled approach is much more sensitive to the quality of the estimates of the effect of its operators than is an interpreted approach which gets feedback after each operation. Furthermore, the SDD-1 algorithm, like Ingres, is a greedy algorithm. It maximizes the immediate gain, never looking ahead and never backing up. Thus it too is distinctly sub-optimal. The algorithm due to Chu and Hurley for generating the solution space, while inherently exponential, does indeed yield the optimal solution. It may be a reasonable approach if access plans are compiled and used many times.

3.5 Distributed Query Compilation in R*

The R* database [Haas 82] like System R [Astrahan 76] on which it is based, takes a query compilation approach to distributed query optimization. R* compiles SQL statements into low level programs or *access modules* which make calls to the *Research Storage System* or RSS* routines. In System R the access modules were actually machine code. In R* they are a higher level representation, similar to Pascal P-codes, which are executed by an access interpreter at run time [Daniels 82]. The access modules are themselves stored in the database. The designers believe that by compiling queries and re-using the access modules, they achieve considerable performance improvements over systems such as Ingres which use an interpretive approach [Chamber-

lin 81]. By compiling queries, R* avoids the overhead of having to perform access path selection each time a query is run. However, R* must record with each access module, the dependencies upon which it is based to ensure that nothing has changed between compilation time and execution time which would necessitate re-compilation [Daniels 82]. For example, an index upon which an access module relies may have been dropped or a user's authorization may have changed.

Because queries are assumed to be compiled off line and used many times, the designers of R* are willing to allow the optimizer do more work to find an optimal access plan than were the designers of Ingres or SDD-1. Thus R* uses exhaustive search to find the optimal solution. The search algorithm considers both nested loop and merge scan methods for performing joins. It also considers all possible orders of table access and all methods available for accessing each table [Daniels 82].

The cost of performing the exhaustive search is reduced by the use of dynamic programming techniques [Selinger 79]. Dynamic programming is useful in selecting the join orders for multi-table joins. N relation joins are implemented as a sequence of two way joins. Two of the relations are joined together, then each additional relation is successively joined to the composite. The cost of performing each join depends on the cardinality, location and order of the tuples in the composite relation and the single relation being joined, and not on the method in which the composite was formed. This satisfies the principle of optimality [Hillier 74]. Thus only the cheapest method of obtaining each interesting ordering of the composite relation must be maintained in the search tree.

The master site selects the global access plan and distributes the individual parts to be executed at *apprentice* sites. The global plan consists of the order of table

accesses and the method to be used for each multi-site join. Each subquery is compiled separately at the site at which it runs. The costs of retrieving tables at the apprentice sites that are used for the global optimization are based on estimates of the access path which the apprentices will use. The global optimizer does not have complete knowledge about all indexes available to the storage sites. R* allows the apprentices to recompile their portions of a query without requiring global reoptimization. [Ng 82] contends that local recompilation often results in a query processing strategy that is still optimal even though access paths relied upon in the original global plan may have been dropped. R* does allow apprentices to initiate global recompilation if relations have migrated or it is determined by heuristics that the plan resulting from a local recompilation would be greatly sub-optimal.

The R* approach to query processing is quite different from SDD-1 and Ingres. The most major difference is that it uses a tree searching approach. Thus it examines the entire solution space and finds the optimal solution given its estimates of result relation sizes. Ingres and SDD-1 use hill-climbing strategies to incrementally modify an initial feasible solution and they are therefore susceptible to local maximums. R* commits itself to an entire access plan before executing any part. Consequently, the R* query optimizer cannot change its strategy if one of its estimates turns out to be very inaccurate. Since it relies less on heuristics which contain knowledge about the environment and more on search, the R* algorithm is more easily adaptable to different cost environments. There is no assumption that network communications are a bottleneck built into the algorithm.

It must be remembered that the R* algorithm is intended to solve a different problem from that faced in Ingres and SDD-1. R* assumes that queries are often complex, reference very large databases, and are used repeatedly. Ingres and SDD-1 are in-

tended for interactive systems in which users sit at terminals and formulate relatively simple queries. Ingres might well benefit from using a compilation and exhaustive search strategy for EQUEL queries.

The next chapter presents qualitative and quantitative results about local networks as an environment in which to do query processing. Chapter 4 reports measurements which determine values for the parameters of the cost model introduced in Chapter 2. Using these values the relative importance of network traffic, local processing and disk retrievals in the cost function is determined.

CHAPTER 4

The Local Network Environment

4.1 The Computing Environment of the Future

It is widely predicted and partially demonstrated that the primary role of the computer in the office of the future will be as a communication system. While "communication system" used to simply simply to the telephone and mail systems, it now carries a much broader meaning. Each function of an office: data storage and processing, document preparation, forms creation and completion, planning, and forecasting must become an integral part of the office system. Much of these functions depend on the data management services if they are to be successful. This thesis attacks some of the problems associated with designing a distributed database to operate in the new generation of computer networks.

The modern computing environment (as opposed to that of the future) is characterized by a large central computer facility with a hodgepodge of smaller systems. The central facility is responsible for maintaining large databases such as employee records and payroll applications, and operates primarily as a batch processor. Because of the inflexibility of the central facility, most large companies have acquired many satellite minicomputer systems. These smaller machines usually belong to individual departments or projects and consequently there is very little coordination among

them. Thus they often cannot communicate with each other without heroic human intervention (carrying tapes), cannot share expensive hardware like printers, cannot share software or data, and run different operating systems requiring the retraining of workers when they change projects. The relatively low price of these minicomputers allows this chaos to come about without any coordination from higher levels of management. However, now that many of these systems are in place, it is increasingly apparent that together, these systems represent an enormous expense and warrant more careful coordination and management.

Into this picture has come a new breed of computer, the *personal work station*. A workstation with a processor of equivalent power to a large minicomputer, a half megabyte or more of memory, a winchester disk, a network interface and a bitmap terminal costs only a few thousand dollars. These machines are designed and priced to devote one machine to each person.

However, a stand alone desk top computer does little more than an electric typewriter and a filing cabinet in helping with the real business of an office, communication. These systems must be tied together by a local area network so that users may send mail, share files, and access expensive shared resources. Thus the local area network is a primary component of the new generation of office systems.

The underlying topology of this office system of the future is finally crystalizing. The model will probably consist of an ethernet like local area network snaking through each building or group of buildings. There will be gateways to other local area networks of similar or different technologies (e.g. ring networks) and also gateways to long distance networks. These long haul networks, primarily provided by common carriers, will link branch offices in other cities together and also provide access to other services

such as market reports, mail systems, and airline reservations. Within the building, each qualified employee will have a workstation on his desk with a small amount of disk storage. There may remain a central facility, also tied to the network consisting of a large mainframe for computation intensive jobs and record keeping. It will be possible for the network to run a distributed operating system that makes the network transparent to most users. The data and computing resources of the central facility as well as all other nodes would be accessible from the individual's workstation in such a transparent distributed system.

4.2 How This Differs From Modern Networks

There are three common models of the modern computer system that are subsumed by the above picture of the distributed system of the future. They are, the large central facility, the loosely connected local network of small machines, and the low bandwidth nationwide network linking local systems. By virtue of their different topologies, user model, and resource cost parameters, transparent local networks make a very different environment in which to perform distributed database query optimization.

4.2.1 Qualitative Differences

The network transparent local area network presents a very different model of the computing resource to the user which affects usage patterns. The user's model of the system is derived more from the single machine model than the network model. The machines exhibit a much higher degree of integration and interaction than in a long haul network. Access of remote data is very much more common than in a long distance network because the user level distinction between remote and local has been blurred.

The motivations for data placement and replication are also very different. In a nationwide distributed database, data is stored near the site where it is most often used because of the desire to minimize access time and communications costs. Data is replicated if efficient access is needed at several sites. As remote access can be almost as fast as local access in a local area network, the motivation to store data where it is most often needed is decreased. Similarly, there is little need to replicate data at many or all usage sites. The primary motivation for replicating data in a local area network database is the desire to maintain data availability in the face of node failures. Data may also be placed by value to decrease not access time, but query processing time. For example, we might fragment the employee relation so that employees on project one are stored on sites one and two and all other employees are stored on site three. We might instead choose to balance the number of employee tuples stored at each site in order to maximize parallelism when processing employee records.

Thus, local network databases differ from long haul nets in a number of qualitative ways which influence query processing in this environment. The data usage pattern is likely to be very different in that remote data is accessed much more frequently than in long distance networks. Data distribution and replication occurs for different reasons in local network databases. Most importantly, because the user model of the system is that of a single machine rather than a network, the database design is oriented less towards conserving the network than towards serving the users.

4.2.2 Quantitative Differences

Most important to this thesis is the way in which local network databases differ quantitatively from widely distributed databases. As we have said before, the conventional wisdom in distributed databases as been that the network communications

bandwidth is the scarcest resource by several orders of magnitude. Consequently, the query processing algorithms have been designed to minimize network traffic at the expense of local processing and disk access. We question the validity of this approach not only in local network databases but in long haul nets as well.

The query processing cost model developed in section 2.3 is designed to illustrate and quantify the resource cost parameters in the local network database. Recall that the cost model was in terms of the constants, P, C, and D which are the processor cost per tuple, the communication cost per page, and the disk retrieval cost per page, respectively. These constants define the query processing cost environment. The total cost to process a query in this model is given by:

$$TotalCost = D * \#pagefetches + P * \#tuplerequests + C * \#messages$$

For example, consider a join between the supplier relation and the spq relation on the S# attribute.

```
RANGE OF s IS supplier
RANGE OF Y IS spq
RETRIEVE (s.all, y.all)
WHERE (s.S# = y.S#)
```

Assume that supplier has 1000 tuples on 100 1K byte pages with an unclustered index on S# with 1000 different S# values and 10 index pages. Supply has 5000 tuples on 200 1K byte pages with a clustered index on S#, 1000 S# values and 30 index pages.

In order to join spq and supplier when they are both located on the same site, the best policy would be to scan the supplier relation sequentially. The system would join each supplier tuple to all spq tuples with a matching S# value obtained by using the index. This requires that the system retrieve do 100 page fetches for the supplier

relation, 1000 page fetches of spq relation data pages, and 30 page fetches for the spq relation index. In addition, 6000 tuple requests are issued by the processor. Thus the total cost is given by:

$$TotalCost = 1130 * D + 6000 * P.$$

If spq and supplier are on different sites, we must add the cost of 101 messages to request service and to retrieve the 100 supplier data pages. The cost equation for this distributed join now becomes:

$$TotalCost = 1130 * D + 6000 * P + 101 * C.$$

In a geographically distributed database, a reasonable ration of C to D to P is 440:20:1 [Selinger 80]. This ratio is obtained by assuming a 50Kb network, a 1 MIP (370/158) processor running R*, and a 3350 disk. For this cost configuration, the total cost is 62% due to message traffic, 30% due to disk access, and 8% due to processing cost.

This cost ratio is based on a slow, Arpanet like network linking relatively fast machines and fast disks. With these parameters, the cost due to the network is approximately double the disk and processing cost. For smaller queries, the dominance of network costs is more pronounced. This is because the communications costs are better than linear in the sizes of the relations (the algorithm can choose to transmit the smaller), while the disk and processor costs are worse than linear. To illustrate this fact, consider joining two relations stored on different machines, each containing only one tuple. The cost is 2C (to send a request and receive the tuple), plus 2D (to retrieve the two tuples), plus 2P to process the two tuples. This yields a total cost which is 95% due to communications. queries. As the query becomes more complex or the volume of data increases, the amount of work required of the database approaches that done by the communications network in this model.

It is important that this model is not considering the queueing delays due to the slow network. A 50 Kb network is indeed a bottleneck. Thus, systems such as SDD-1 working in this environment choose to minimize only the volume of communications.

The relevant question being addressed in this thesis is how this environment changes in high speed networks where communications are not a bottleneck and queueing delays on the network are negligible. The communications media in our local network is about 200 times faster than the one modeled above and the processors in our network operate at about one half MIP.

Measurements were done of the Ingres database running on Vax 11/750s with Fujitsu Eagle disks connected by a 10mb ethernet. Appendix 1 gives the details of how measurements were obtained. The results of the measurements are that the ratio of C to D to P is 8.5 to 13.6 to 1. For the distributed join in the above example, the total cost is 69.1% due to disk page fetches, 27% due to processing tuples, and 3.9% due to the use of the network. It should also be noted that this example uses indexes which greatly reduce the disk access requirements. Without the indexes, the system would have to perform a more expensive kind of join either using a nested loop or creating an access path. For example, Ingres would perform tuple substitution, joining each tuple of the supplier relation with the entire spq relation requiring 200,000 page fetches of the spq relation and the processing of 500,000 tuples while still sending only 101 pages over the network. Thus, in the above query which is a worst case example, only 3.9% of the work resulted from the presence of machine boundaries. Even in cases where disk access and processing are minimized by the existence of efficient access paths, the cost of using the network is insignificant.

4.3 Implications for Query Processing

Query processing algorithms generally attempt to minimize either total time, response time, or a linear combination of the two. Minimizing total time maximizes system throughput, but it does not give any incentive for the algorithm to use parallel processing to speed an individual query. The right solution seems to be to minimize a weighted combination of total time and response time. The weights should be variable so that the database can favor response time when the system is lightly loaded and total time when it is heavily loaded. What to minimize is a policy decision and the database should be capable of adapting to a changing policy.

As was stated in chapter 2, this total cost function is not an optimization function. This is because the units are seconds, and seconds of network time are not the same as seconds of disk or processor time. It would be a rather arbitrary rule just to minimize the sum of these three times. It does, however, make sense if we weight time on the disk, time on the net and time on the processor in some way so as their value is equivalent.

A good alternative for weighting the relative costs of these resources would be to weight them in proportion to their dollar costs. This calculation is done in Appendix 1. The results are that the ratio of the cost of one second of network time to one second of disk time to one second of processor time is 1 to 7.5 to 19.4 and the dollar based ratio of C to D to P is 8 to 6.8 to 1. For our sample join with two sites, this yields a total cost which is 53% due to page fetches, 42.5% due to processing, and 5.7% due to communications. The contributions of processing cost and disk cost are both an order of magnitude greater than message cost.

The primary result of these measurements is that disk retrieval and processing costs are significantly more important than communications costs in a local network environment. What are the implications of this result for query processing? First, we would like to perform the expensive tasks in parallel. By decomposing a task into pieces which may be executed in parallel, the time required to complete the task can be reduced to the maximum of each of the pieces. The most expensive tasks are first, retrieving the data from disk, and second, the local processing. Retrieving the data from disk in parallel can only be achieved by storing data on different sites or different disks. Since network communications are so cheap, it is cost effective to fragment data at many sites in order to retrieve and process in parallel, even if the usage site is known in advance. These results suggest that communications may be thought of as free or instantaneous by the optimizer without serious effect on the efficiency of the resulting schedule.

Secondly, this research underlines the necessity to maintain efficient access paths for relations. Direct access indexes greatly reduce the number of times a given page is retrieved from disk. For example, to solve the above sample query required 1130 page fetches. If there were no index on the spq relation the system would have to use the index on the supplier relation requiring 5210 page fetches. If there were no index on the spq relation index however, the system would either have to create an access path dynamically, sort the relations, or scan the spq relation for each tuple in the supplier relation. Ingres uses tuple substitution (nested loop) for all but the final join (when it may do a merge join); and thus, has no choice but to scan the whole spq relation for each tuple. This requires 200,100 page fetches, none of which may be done in parallel. The bottom line is that, just as in single machine databases, indexes are essential to reasonable performance.

A corollary to the above is that the system must avoid making indexes obsolete due to fragmentation. In a desire to achieve parallel processing, Distributed Ingres has proposed the fragment and replicate strategy for distributed query processing [Epstein 78]. This method dynamically fragments one relation and replicates all others. However, in so doing, it renders useless any indexes that may have existed for any of the relations. It has been suggested before that fragment and replicate strategies are only good if the database is already highly fragmented [Sacco 81]. This was because of the communications overhead of fragmenting one relation and replicating all the others at the processing sites. In the local network environment, we have shown that the communications overhead is minimal. However, the need to recreate or do without indexes is catastrophic.

Finally, it is very important to have the pages of the relations well laid out on the disk. Unix and LOCUS, on which Ingres runs, perform read ahead in order to speed the sequential scanning of files. This is particularly useful in database applications where most access is sequential. Unix is not particularly well suited to such applications because it has no provisions to keep sequential blocks of a file colocated on the disk. Though the free list is initially laid out contiguously (blocks placed consecutively separated by enough space to account for interrupt latency), as files are added and deleted, consecutive file blocks tend to become scattered over different cylinders requiring large seeks between data transfers. The small, 1024 byte block size further limits the file system's throughput. As disk throughput is the limiting factor in local network databases, improvements to the file system would greatly improve the performance of databases in Unix. [McKusick 82] suggests several ways of improving the performance of the Unix file system which will soon be incorporated into Unix and LOCUS. They include use of a larger, variable block size, and optimal placing of con-

secutive disk blocks. [McKusick 82] predicts performance gains of up to 10 times over the old Unix file system. Any gain at all would be directly reflected in database performance.

This chapter has described how a local network differs from a long distance network as an environment in which to do query processing. We concluded that they differ in degree of site interaction and data sharing, user model, motivations and pattern of data distribution and replication, and in cost model parameters. Measurements conducted on our distributed Ingres running over a 10mb ethernet indicate that the cost of network communication is indeed negligible compared to the costs of local processing and disk access.

The next chapter examines possible database architectures. In particular it describes the facilities provided by the LOCUS network transparent operating system, which make it a good foundation on which to build a distributed database system.

CHAPTER 5

Case Study, A Distributed Database in LOCUS

5.1 Conventional DDMS Architecture

Unlike the operating system community, the database community has generally agreed from the start that distributed databases should present the user with a transparent view of the data. Transparent, in this context, implies that the user does not have to know where the data is physically stored either on the disk or in the network. In particular, there is no notion of local or remote; resources are accessed the same way regardless of their storage site. A relation name resolves to the same set of data, independent of the site at which the name is uttered, and a name does not imply a particular storage site. Data or users may move to different sites without affecting access methods. From the user's point of view, there is nothing distributed about the database except the volume of data that is accessible. Very few systems that are actually functional go very far toward meeting these goals.

If the user is to view the distributed system as a single site database, then all of the problems of data distribution must be handled by some component of the system. These problems include maintaining concurrency of multiple copies of data, coordinating shared access to data, recovering from partial failures, distributed catalog management, and distributed query processing. In most distributed databases designed to

date, all of these functions are handled explicitly by the database. A typical distributed database architecture is described in [Hevner 79] and is shown in figure 3.

When a query is submitted to this system, the *user interface* forwards it to the *query processing subsystem*, which is responsible for implementing the optimization algorithm. The query optimizer must interact with several other modules. It is the job of the *integrity subsystem* to determine a unique materialization of the database when replicated relations are involved. The integrity subsystem also implements the update synchronization functions which insures the consistency and integrity of the database in the face of concurrent queries. Once the optimizer has received the materialization information, it determines the access plan for the query and passes it to the *scheduler subsystem*.

The role of the scheduler is to send the commands to the local database systems on each site. At the site originating the query, the scheduler coordinates the distribution of the pieces access plan to the cooperating sites. At each of the other sites, the scheduler receives the network messages, forwards commands to the local database, and sends the results back to the master site.

The *reliability subsystem* monitors the network and other sites. It makes the database more reliable by sensing when a communications line or node that is participating in an active query fails. It deals with the failure by backing out the query, and possibly restarting it with a different copy of some part of the data. The reliability subsystem is also responsible for reintegrating a site or group of sites back into the network when failed components return.

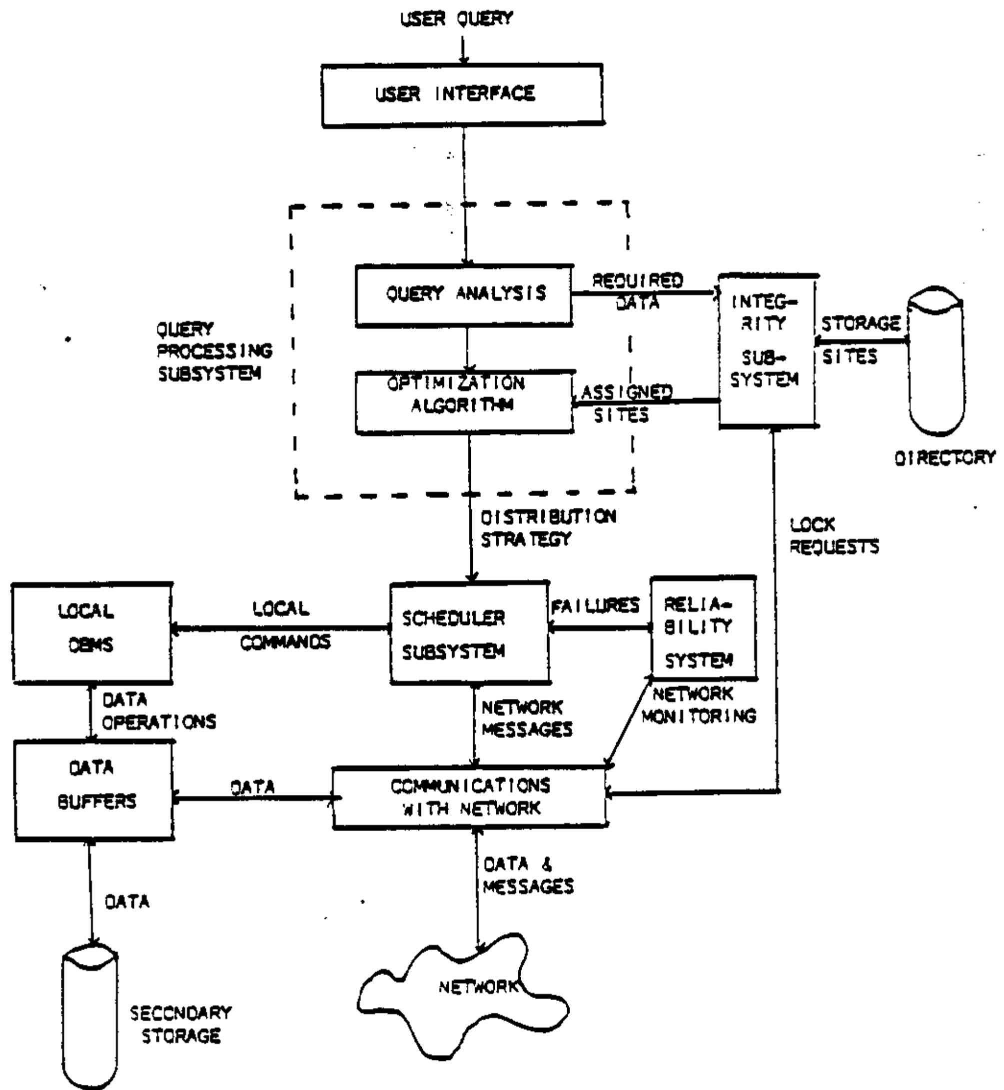


Figure 3: Query processing in a distributed database system

Correct implementation of these subsystems in a distributed environment is very expensive. The integrity subsystem must implement either a locking or timestamp protocol to coordinate concurrent access to data and be able to detect or prevent deadlocks caused by the protocol. The reliability subsystem must be able to "undo" the effect of a query or part of a query in the case of a failure of a node or a communications link during execution. A log or shadow page mechanism must be used along with a two phase commit protocol to guarantee atomicity of updates. It is these protocols which deal with the distributed nature of the database that make building distributed databases so expensive and difficult.

5.2 Two New Architectures

The network transparent application program interface of the LOCUS network operating system permits a much simpler method of creating distributed databases. As has been observed earlier, many of these protocols that deal with the network are needed by many applications in the distributed environment. At therefore makes sense to implement them once in the operating system and to make them available to all programs.

This thesis examines implementation of a distributed database by pushing the integrity subsystem, reliability subsystem, and communications subsystem, below the level of the operating system interface. Two new distributed database architectures have been implemented and tested. In the first model, hereafter called the "trivial" model, a distributed database is obtained by running a standard Unix single site database in the LOCUS system. The location of the data is not visible to the database in this model. The database runs exactly as if it were on a single machine. When it issues an open call for a file that is not local, the operating system takes care of choosing

a storage site to supply the file, and bringing the pages across the network. The single site database becomes trivially distributed without adding one line of code or even recompiling. The database thus obtained is capable of retrieving data from remote sites, transparently replicating data, continuing to operate in the face of partial failures, gracefully recovering from such failures, and maintaining the consistency of data while allowing concurrent access on different sites. This trivial database, obtained almost for free, gives us much of the desired function of a distributed data management system. The performance of Ingres running in LOCUS and retrieving remote data is reported in section 5.4.

While it is adequate for many applications, the above model does not exploit the network to its fullest potential. It does take advantage of the existence of multiple sites to increase the reliability and availability of the data by allowing for replication on many sites. However, it wastes the potential for greatly improved performance provided by the extra processors in the network which can work on a query in parallel. In order to take advantage of the potential for parallelism, the query optimization algorithm of the single site database must be extended to know about the additional processors. In addition, a scheduler subsystem which was not needed in the first model must be constructed.

The second model of the distributed database consists of a local database management system, a simple scheduler subsystem, and a query processing mechanism. The local database management system is basically the same as that found in the first model. It is slightly modified to communicate with the scheduler subsystem when remote actions must be initiated. The scheduler subsystem performs the functions described above in the section on the conventional database architecture. However, its construction is very much easier in a LOCUS system because the database does not

have to implement its own interprocess communications system. The operating system provides network pipes [Kirby 82] with which scheduler systems on different sites can send instructions and receive results. Furthermore, it does not have to implement the "move relation from site A to site B" primitive found in most distributed databases, nor keep track of the location of database information, because LOCUS already makes all files available on all sites and maintains catalogs.

This distributed database architecture gives us a fully functional distributed database. Any query processing algorithm can be implemented in the optimizer and executed by the scheduler. The scheduler can cause operations to happen in parallel on remote sites and wait for these remote operations to complete.

5.3 Implementation

In order to experiment with query processing in a local network and to investigate the premise that a full distributed database can be easily constructed out of a single site system, a distributed query processing mechanism was added to Ingres. This mechanism allows the master Ingres process at the query origination site to cause subqueries to be performed at any or all remote sites in parallel. The structure of this mechanism is shown in Figure 4.

Remote subqueries in this system are performed by Ingres server daemons called iservers. Each machine creates one or more iserver processes when it is booted. Also created at boot time are two network pipes stored on that machine. The iserver process immediately attempts to open the first pipe named "remotepipe", for read. The process hangs waiting for some other process to open the other end of the pipe for write.

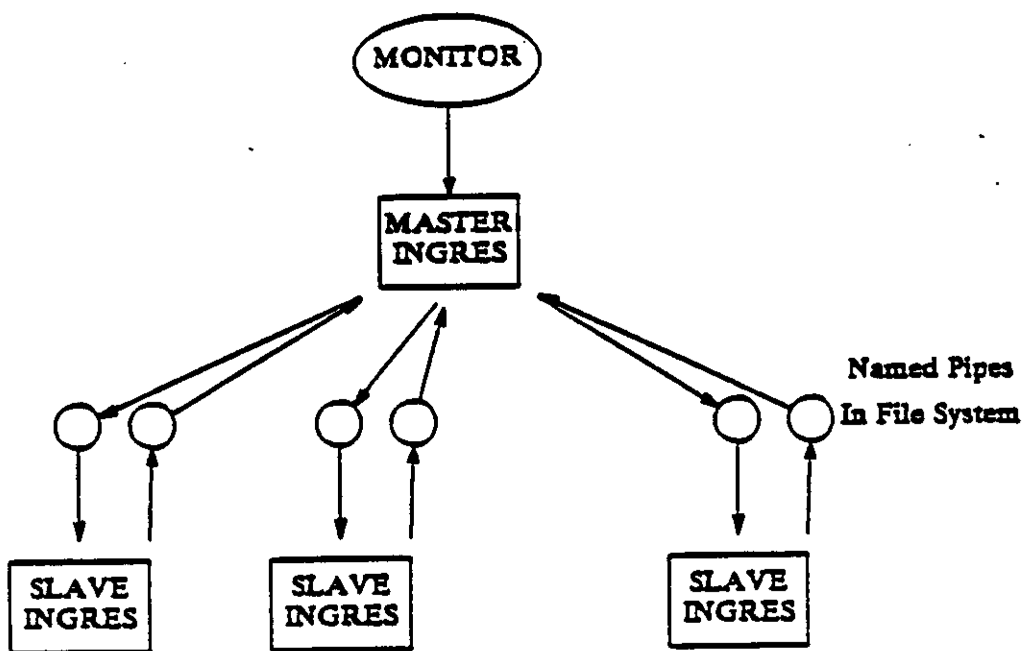


Figure 4: Structure of the remote subquery mechanism

When a user wishes to run Ingres, he forks a monitor and a master Ingres process at his site. If the master Ingres needs to run remote subqueries, if opens the "remotepipe" and "respipe" named pipes for write and read, respectively, for each of the sites to be used. The first open wakes up the iserver processes on each of those sites. The iservers in turn open for write their respipes which they use to send results back to the master Ingres. The master first sends the necessary control information to the iservers. This information includes the name of the database, the pathname of the Ingres subtree in the file system, and the debug variables to turn on.

When the query optimizer in the master Ingres has a subquery that it wishes to execute remotely, it calls the OVQP module which implements the scheduler function. The scheduler receives a pointer to the parse tree of the subquery and a list of sites at which to run the fragment. As there is currently no optimizer for this implementation, the query processing strategy must be controlled by hand. The default strategy is to run the query using the standard Ingres decomposition method with one variable queries run at the storage sites of the data.

The parse tree of the subquery is converted back into Quel to be sent to the servers. Using this high level interface over the network causes some small performance penalties, but has some important advantages. Because they take Quel as input, the iserver modules are very similar to the main Ingres processes in standard Ingres; and consequently, required only a small amount of programming time to construct. Furthermore, any type of database can be imbued with the ability to convert its inner data structures which represent subqueries into Quel. In such a way, this mechanism allows heterogeneous databases to request service from the slave iserver processes. Similarly, slave processes can be constructed with Quel parsers to provide access by distributed Ingres databases to data stored in other types of databases. The

high level, standardized interface also provides different sites the individual sites the local autonomy to run different releases of the system. There is some overhead in recursively traversing the tree to convert it to the original Quel and parseing it again at the service sites. This overhead is small as the subqueries that are actually executed are generally simple queries, usually involving only a single relation. The ease of implementation and the potential for heterogeneous databases made translation of subqueries into Quel desirable.

The query is written to each of the remotepipes. The iservers read the query, parse it, and execute it locally. There is currently no provision for the iservers to cause other remote subqueries to be run. If the local subqueries still require remote data, LOCUS provides access to it transparently as if the file were local. Thus, the query is globally optimized at the master site. The iservers run the standard Ingres decomposition algorithm on the query to do the local optimization. The iservers are able to put the results of the subqueries into temporary relations or send them, tuple by tuple, back to the master via the respipes. The master site waits on each of the respipes in turn to read the result data.

Because work on concurrency control in LOCUS is not yet completed, concurrent access to the system catalogs by the remote iserver daemons could not be permitted. To avoid having the many servers accessing the system catalog relation catalogs concurrently, the catalogs were replicated at each site, and updates to the catalogs were temporarily avoided. Using the LOCUS hidden directory mechanism, the same relation name issued at different sites was forced to map to the local copy of that relation, avoiding concurrent access between processes on different sites. This has avoided rather than solved the issue of the design of a distributed relational catalog. The tools that LOCUS should provide and the design of a catalog mechanism for a distributed

database in local network is itself a good topic for a thesis.

A simple implementation of horizontally fragmented relations was also achieved in a similar manner for the purposes of performance testing of parallel access to fragments. This was done by creating the database in the directory /usr/database. The /usr directory was in fact a hidden directory containing components for each of the sites in the network. Mounted on the component of each site was a local version of the file system. The same file name, when opened by the database on different sites, mapped to that file in different versions of the /usr file system. For example, test queries were run retrieving data from the supplier relation, which contained a total of 30000 tuples. The relation was actually stored in three files on three different sites. Supplier was fragmented with one-third of its tuples being stored on each of the three sites. Hidden directories appear to be an appropriate tool for building fragmented relations in a transparent file system.

Currently, while an iserver daemon is performing a remote subquery, it keeps open the "remotep" network pipe. If another master Ingres running anywhere else in the network requests service from that site, the second master's open of the pipe will fail until the first subquery is finished. Clearly this is unacceptable behavior in a multiuser environment. There are very simple changes to the mechanism as it exists, which would alleviate this problem. One solution would be to have the iserver daemon fork a child process to service each request. The parent could then go back to waiting on the pipe for further requests. This method incurs the overhead of doing a local fork to create a new process. Similar overhead is incurred by the use of an exec of the slave by the master across the network. Alternatively, several daemons could be created for each site when it is booted, each waiting for requests on its own network pipe. Master Ingres processes requiring remote slave service would keep trying different pipes for the

desired site until one is found which is not already open. A site can control the number of remote queries it is willing to serve at a time by the number of daemons it creates. This second solution appears preferable.

This implementation took about five man months to complete with only about two months devoted to coding and testing. The bulk of the execution time was spent in understanding the way standard Ingres works in order to be able to make modifications. A programmer who is experienced with the system he is modifying, could convert much of a single site database into a distributed one in the method described above in much less time. The remote query mechanism required only about 1000 lines of code and modifications to about 25 Ingres source files. The bulk of the new code performed the translation of the parse tree back into Quel. The remaingin code changes occurred mainly in the high level control routines, and the One Variable Query Processor.

The paucity of VAX Ingres documentation contributes to the implementation difficulties. The existing *Design and Implementation of Ingres* [Stonebraker 75b] document describes Ingres as it was originally created in 1976 for PDP 11 series computers. Due to the limited address space of these processers, Ingres was implemented as a chain of cooperating processes connected by Unix pipes. The original database system has since been modified to run on VAXs as a single process. However, the relics of the multi-process implementation remain within the source code.

In order to map the five processes into a single process, VAX Ingres is implemented as a table driven system with its own context stacking mechanism. The main routine is invoked with an initial state. This state is mapped to a function which the database must perform by an internal table. Also contained in the table is the next

state that should be entered after the current one is completed. When a new state or function is entered, the structure containing all local variables is placed on the context stack. If after completion of a function, there is no corresponding next state in the table, the context stack is popped and execution of the previous state is continued. This mechanism is well suited to performing the recursive query processing algorithm shown in Chapter 3, Figure 2.

Both the master Ingres process and the iserver daemon take Quel queries as input from a pipe. Consequently, both have the Parser function as their initial state. These two processes share a large amount of code and function. Differences in their behavior are controlled by checking the global flag, Remq, which indicates whether the process is a master or a remote slave. The Parser calls view, integrity and decomposition functions, which in turn cause one variable subqueries to run. Gaining initial understanding the process table and the context stacking mechanism is time consuming, as they make tracing the flow of control of the system difficult. However, once understanding is attained, modifying Ingres is not exceedingly difficult.

5.4 Performance Results

Timing measurements were performed to determine how this implementation of a remote query processing mechanism compared to the various other incarnations of Ingres. For these measurements, the following queries were run:

```
Q1:   RANGE OF t IS test
      RETRIEVE (t.all) WHERE (t.number < 0)

Q2:   RANGE of t IS test
      RETRIEVE (t.all)
```

The relation test contained 30000 tuples, each 100 bytes long so the relation contained a total of 3MB of data. No tuple had a number field less than 0 so no tuples satisfied the qualification in the first query. The second query had no qualifications; and thus, all tuples were returned. In both cases, the output of the query was directed into /dev/null so the cost of scanning the entire relation was measured, while the cost of printing the results on the terminal was not. These queries were run on standard Ingres running on LOCUS with purely local data, on distributed Ingres with local data, on standard Ingres with all data on a single remote site, on distributed Ingres with all data on a single remote site, and on distributed Ingres with the data fragmented between three remote sites. The average startup and shutdown time of 7 seconds and 8 seconds for standard Ingres and distributed Ingres respectively have been subtracted from the elapsed time so that it reflects the true time required for a single query.

	Q1	Q2
Ingres with local data	2:15	4:00
Dist. Ingres with local data	2:19	4:01
Ingres with remote data	3:23	4:33
Dist. Ingres remote data	2:25	4:22
Dist. Ingres data on 3 sites	:50	2:09

Table 1: Performance of different Ingres architectures

The results for the first query are approximately what we would expect. It takes only slightly longer to run distributed Ingres for local data than to run standard Ingres. The difference is the time it takes to parse and unparse the query, and to handle the synchronization over the named pipe. It would not be difficult to force the the master Ingres process to go ahead and run all queries on local data itself, to avoid this

penalty when all data is local. The time required to run the standard database on remote data is about one minute eight seconds longer than for local data. This is quite reasonable, since the time required to read a 3MB file across the network in LOCUS is about one minute five seconds. Running distributed Ingres on remote data with the slave process running at the storage site takes about the same amount of time as did the first two trials. The time required to send control messages through the network pipe accounts for the few extra seconds. Running distributed Ingres with remote data is about one minute faster than is standard Ingres, because the distributed version is able to run at the storage site avoiding any data being sent over the network. Retrieving the data in parallel from three sites runs as expected in about one third of the time that retrieving the entire relation from one site does. This final result is very encouraging as it implies that for relatively large queries, the overhead of the remote subquery mechanism is amortized and we do achieve the performance gains of parallel operation.

The only difference between the first query and the second is that in the first, no tuples were returned, while in the second, all tuples satisfied the request. For standard Ingres with local data, about 1 minute 45 seconds of overhead was added by the extra processing that outputting the tuples required. When the data is on another site, 33 seconds of additional time is required by standard Ingres. Distributed Ingres requires about the same time as standard Ingres when the data is local to the process. When distributed Ingres is run with the data on a remote site, 21 seconds of elapsed time is added by sending the data across the network. The three site case performed very well, completing in under half the time of the single remote site trial. The full factor of three speed up that was achieved by parallel processing in Q1 is not quite realized here due to inefficiencies in the remote pipe mechanism. The performance of the remote query mechanism for Q2 is again very encouraging.

These measurements have revealed some room for optimization in the design of the remote named pipe mechanism in LOCUS. The contents of a pipe is sent across the network each time the reader is ready for more data. If the reading process is consuming the data faster than the writer can produce it, very poor network utilization may result. As it was first constructed, the code for returning result tuples to the master site read all tuples from each of the slaves in turn. The three site trial of Q2 required 7 minutes 34 seconds to complete. This code was subsequently changed so that the master reads one block from each of the slaves each time through the loop. This gives each of the remote daemons time to fill up more of the buffer before it is pulled over the network by the master process, greatly reducing the number of network messages being sent for each page of data. This changed improved the time from 7 minutes 34 seconds to 2 minutes 59 seconds. Then the size of the buffer which the slave writes to the network pipe was increased from 128 bytes to 1024 bytes. This final change improved the time to the reported 2 minutes 9 seconds. Careful optimization of the remote pipe mechanism within the operating system could free application programs from the need to be aware of these sorts of performance considerations.

5.5 Comparison With Berkeley Distributed Ingres

Implementation is currently underway at the University of California, Berkeley of Distributed Ingres based on the 4.1cBSD version of Unix [Stonebraker 83a]. It makes use of enhancements to Unix made at Berkeley including remote interprocess communications and remote process execution. The design of Berkeley Distributed Ingres is in many ways similar to that executed here with a master Ingres at the query origination site and slave processes at the data storage sites. The Berkeley version uses a lower level interface between the master and slaves so the slave processes need no parser. Rather more work at Berkeley has gone into the user interface to fragmented

relations and into concurrency control but, the function provided is very similar to that provided by this thesis.

Valid performance comparisons between Berkeley Distributed Ingres and LOCUS distributed Ingres are difficult. Berkeley's published benchmarks were run using a network with a 11/780 and several 11/750s connected with a 3 mbit ethernet. The measurements reported here used a 10 mbit ethernet and only 11/750s. Berkeley's implementation, like that reported above, required about the same amount of time as standard Ingres when data was local. They did suffer some penalty when data was remote. The Berkeley system was also tested using three slaves in parallel. This trial achieved only a slight improvement in response time over the single site test. Running Berkeley Distributed Ingres on a single site with the same amount remote data took 2:54 of elapsed time, while the three site case took 2:37. Thus, only modest gains were made by the use of parallel processing. Before any meaningful conclusions can be drawn about the relative performance of these two versions, much more careful testing must be done running both systems in the same environment with the same queries on the same data.

Conclusions may, however, be drawn about the way in in which the two systems were constructed. The Berkeley Distributed Ingres was built in a standard Unix environment enhanced with some remote interprocess communications tools. Consequently, it had to deal internally with access to remote data. To get an initial system up and running took about four man years of programming effort [Stonebraker 83b]. Getting a vaguely equivalent system running in the network transparent environment took only about two man months of programming. While neither system is a fully functioning distributed database, the time required to get a first cut operational is very important, as further enhancements may be made and tested incrementally. That a

system of roughly equivalent function and performance can be built in LOCUS in years less time is a significant result.

The mechanism built for this thesis falls short of being a fully functioning distributed database in several ways. It does not implement a true distributed query optimization algorithm. Its default mode of operation is to decompose the query using the standard Ingres decomposition code and then run the individual subqueries submitted to the One Variable Query Processor on the site which stores the data. Any variation from this default must be hand coded. Implementing a given algorithm should require a few man months of programming. However, what is really required is the implementation of and experimentation with a number of algorithms, a task worthy of a dissertation.

The problems of concurrent access to the database from different sites has not been addressed. It is anticipated that new versions of LOCUS shortly forthcoming will provide the concurrency control tools that will make solving this problem in the database very easy. For the moment, the database is completely locked by the first user. Once suitable tools exist in LOCUS, a few man months of effort would provide a usable mechanism. However, databases really require finer than file granularity locking, and the design of this feature for LOCUS is, again, worthy of a dissertation.

The design of a catalog mechanism appropriate for this environment has not been completed. Such a mechanism must support replicated and fragmented relations and their distribution criteria. The work reported in [Theil 83] begins this task, but has not yet been integrated with the implementation reported here.

Finally, a clean user interface to the defining and accessing and updating distributed relations must be designed. While access to the database is generally network transparent, the user must have some way to specify site dependent information in the data definition language. Implementation of the set of extensions suggested by [Stonebraker 77] could be completed in a few man months of work. A usable distributed Ingres in LOCUS is probably one and a half man years of work away. The solution to all of the problems of a distributed database in a local area network requires breakthroughs equivalent to several dissertations.

In contrast, Berkeley Distributed Ingres requires about ten man years of work beyond the initial version to create a usable distributed database [Stonebraker 83b]. The reason for this difference is that much of the mechanism required to make a distributed data management system has already been built into the LOCUS operating system. Crash recovery, conflict detection, and atomic transactions are examples of features already present in LOCUS and usable by a database. Systems built on top of standard Unix operating systems must provide these expensive functions internally at enormous expense. All results so far have indicated that these features which deal with distributed data can be provided by the operating system, and can be made to perform efficiently enough for database applications.

This implementation of a local network database in LOCUS realizes the goals it set out to achieve. Because of the network transparent environment provided by LOCUS, the time required to build this mechanism was quite short compared to the development time for most distributed systems. While it identifies a necessity for optimization of the network named pipe algorithms, the performance of this mechanism for local, remote, and parallel operations is very encouraging.

CHAPTER 6

Conclusions and Future Research

There are as yet no fully operational distributed relational databases for local networks. Consequently, work in this area has been mostly theoretical, based on experience in geographically distributed systems, or on simulation and modeling. Based on the current trends toward decentralized computer systems and the need for data processing capabilities in office systems, there is little doubt that distributed databases designed for local area networks will be produced in the very near future. This thesis has provided some necessary results pertaining to the problems of query processing and database architectures in local area computer networks.

6.1 Query Processing

This research has provided empirical results about the nature of local networks as an environment in which to do distributed query optimization. We have constructed a cost model with which to compare the cost characteristics of different configurations of machine, network, and disk speeds. Measurements were conducted to determine reasonable values for the costs of local processing, disk access, and network usage. These cost parameters were used to analyze several typical queries. The model showed that for these queries the disk retrieval and processing costs dominate network message costs. In no case was the percentage of the total cost due to the network

greater than 10%.

The quantity that should generally be minimized by the query optimizer is the response time for a query. While in the long haul case this goal was ostensibly equivalent to minimizing network messages, this is not the case in local networks. In the local network environment, the response time is a much more complicated function of the data retrieval, processing and transmission schedules. The reason the function is so complex is that it is very difficult to determine just what actions occur in parallel. There is a significant amount of processing involved in both sending network messages and retrieving data from the disk. These operations do not occur completely in parallel with processing tuples. Furthermore, since the operating system is doing read ahead both on the disk and across the network, it is difficult to tell when the database will have to wait for the data to be available, and when it will already be in the system buffers. Further work is necessary to derive a viable optimization function.

These results indicate that it is reasonable to assume that intersite communications are free. It is far more important that joins be done in the correct order and with the correct inner and outer relations than that they be done at the site which minimizes communications. Furthermore, it is easy to use rules such as those given in [Chu 79] to make decisions about where local processing should be done. It is easier to build an optimizer with an evaluation function which ignores communications costs while avoiding bad processing site choices with rules and heuristics than it is to include the delay due to network messages in the response time function and to search this large space.

Since disk retrieval is the costliest component of query processing, it is important to use all available tactics to reduce the number of page fetches and to increase

parallelism. This is mostly done at database creation time rather than dynamically. These techniques include constructing indexes on commonly referenced attributes, horizontally fragmenting large relations between several sites, and placing the blocks on the disk so as to minimize arm movement.

6.2 Distributed Databases in LOCUS

The second goal of this thesis was to address the difficulty of developing software for a distributed system. Such software is more expensive to build because of the different and more complex interfaces to remote resources, the more complicated model of the underlying system, the richer set of failure modes with which the software must deal, and the desire to mask these problems from the user of the system. This thesis has proposed and demonstrated that a distributed database could be implemented quickly and efficiently using the distributed environment provided by LOCUS. The first implementation involved simply running a standard single site Ingres database on top of a LOCUS network. The distributed database thus obtained could retrieve and update data stored on any site in the network and was resilient to partial failures in the system. This provided much of the desired function. However, it did not address the potential for greatly improved performance that the presence of redundant processors in the net offered.

This first model was extended by a mechanism that allowed concurrent subqueries to be run in parallel at many sites. This mechanism was relatively cheap to build and achieved the potential for parallel operation. Measurements showed that indeed three sites working in parallel could complete a query in slightly over one third the time required by one site. Most distributed database projects ever attempted have devoted tens of man years to reach this level of function. The approach of taking a

single site database, which itself has many tens of man years of work in it, and extending it with some simple mechanisms which have very modest knowledge about the network is shown by this thesis to be a viable and cost effective method of producing a distributed database.

6.3 Future Research

The next step in this research is to build an distributed query optimizer which makes use of the remote query mechanism constructed for this thesis. This is potentially a large piece of work involving many theoretical and experimental results as well as volumes of code. The remote subquery mechanism is general enough to be useful to any type of optimizer; one that does dynamic optimization, or one the compiles queries completely before executing any subqueries. The optimizer can be constructed as a filter which modifies and annotates the parse tree. It then would submit partial subtrees in the case of a dynamic optimizer, or a complete annotated parse tree in the case of a query compiler, to the scheduler module. This design facilitates experimentation with different optimization algorithms because one optimizer can be substituted for another as long as it adheres to the same input and output interfaces. Further work in query processing in this environment may address the use of dynamic programming techniques to determine optimal processing and storage sites in the presence of replicated data, handling distribution criteria which reference multiple relations, and dealing with vertically fragmented relations.

There are deeper questions about the relationship between an operating system and a database. Traditionally, database systems have not been built on top of operating systems, but rather beneath, or instead of operating systems. This work has shown that databases can be constructed much more cheaply on top of a distributed tran-

sparent operating system. However, it has not been proven that this architecture can be made to perform competitively with systems such as IMS, which perform disk management and processor scheduling themselves. It is possible that by implementing more database functions deeper within LOCUS much greater performance could be achieved. For example, the relational storage structures and access methods could be implemented within a strongly typed file system. These are questions which must be addressed, as there is no doubt that the coming generation of distributed office systems will be called upon to provide both operating system and database services.

APPENDIX 1

Determining Cost Ratios

We wish to determine appropriate values for D, C, and P, the time based costs of retrieving a page from disk, the cost of sending a page across the network, and the cost of processing a tuple, respectively.

The time required to fetch a page from the disk must include the seek time and average rotational delay of the disk, the time to transfer the data, and the time required to process the read request and disk interrupt. The disk parameters were obtained from the Fujitsu manual [Fujitsu 79]. The processing time was obtained using the measurement software developed by Arthor Goldberg [Goldberg 82]. The components of disk access time are summarized in Table 2 with all times shown in milliseconds.

Component	Time
Average seek time plus rotational delay	25.58
Read call processing	4.25
Disk interrupt processing	1.00
Transfer time for one 1024 byte page	.54
Total page fetch time	31.37

Table 2: Disk Costs (in milliseconds)

The time based cost of using the communications network is rather more complicated to measure. This implementation of a distributed database uses the remote pipe mechanism of LOCUS to perform interprocess communications between cooperating sites. Data is pulled over the network by the *read system call*. The operating system sends a *request message* to the service site and waits to receive a *response mes-*

sage. The arrival of the request message at the remote site causes an interrupt. The interrupt handler places the message on a queue to be processed by a *server process*, which eventually generates a read response output interrupt to send back the requested data page. The interrupt handler sends the data page across the network to the original site where a read response input interrupt is generated and the requesting process awakened. Thus, the communications cost measurement must include the propagation delays on the network, the time to process the request on both sites, and the time to handle the network interrupts. The results are summarized in Table 3.

Local read request processing	4.26
Read request output interrupt	1.6
Read response input interrupt	3.2
Total local processing	9.06
Remote server process time	5.33
Read request input interrupt	2.2
Read response output interrupt	1.9
Total remote processing	9.43
Transfer time on 10mb ethernet	1.0
Total message time	19.49

Table 3: Communications Costs in milliseconds

To determine the processing cost per tuple, time calls were placed in Ingres around the loop which scans all tuples in a relation sequentially. Ingres was modified slightly to force it to process a single tuple 10,000 times. This was done so that we would be guaranteed that the tuple was in core so that we were not measuring any disk access time. However, the processing time per tuple depends, to some extent, on the type of processing being done. For example, processing a tuple may involve checking zero or more of its domains against a list of qualifier clauses. Table 4 shows the time required to process a tuple for zero through four qualifier clauses. For this test, a

32 byte tuple with 3 domains was used.

	#clauses				
	0	1	2	3	4
time (ms)	2.0	2.3	3.0	3.5	4.2

Table 4: Processing time versus number of clauses

Tests were also run to determine whether or not the processing time depended heavily on the length of the tuple or on the number of attributes in the tuple. Table 5 shows the processing time in milliseconds versus the length of the tuple in bytes for a query on one integer domain. Table 6 shows the processing time versus the number of attributes in the tuple.

	bytes			
	4	8	16	32
time (ms)	3.1	3.1	3.2	3.4

Table 5: Processing time versus length of tuple

	#attributes				
	1	2	3	4	8
time (ms)	3.4	3.7	4.1	4.5	4.8

Table 6: Processing time versus number of attributes

These measurements show that the processing time depends primarily on the number of clauses in the qualifications list. Since we are primarily interested in join optimization in this thesis, and single attribute equijoins are by far the most common type of join, the 2.3 millisecond measurement is the correct value for P. This implies that Ingres performs approximately 1400 instructions per tuple.

We now have a good idea of the relative time based cost of using the processor, disk, and ethernet. The ratio of C to D to P is 19.49ms. to 31.37ms to 2.3ms or, more simply, 8.5 to 13.6 to 1.

If we wish to compare more fairly the cost of using these resources for a given amount of time, we must factor in the "opportunity cost" of each type of resource. To do this we want the cost per second of using each resource. We have already shown that, for example, a page fetch requires approximately 26ms of disk time and 5ms of the processor's time; thus, we can get the real cost of performing a page fetch. Table 7 shows the cost of using each of the three resources. The cost for the Ethernet hardware is a per site cost. If we are considering a 10 site network, we must multiply the cost per second by 10 since any site transmitting has sole use of the network and is, in a sense, using all of the network interfaces. However, the portion of network message cost due to the use of the network itself is so small compared to the cost of the processing required, it does not effect the final cost ration. The Ethernet cost includes an Interlan network interface, a transceiver, and 50 feet of cable installed. The disk cost includes the cost of the disk drive and one controller.

Item	Purchase Price	Cost Per Second
Ethernet	$1875.00	$.0000175
Fuji Eagle	$14000.00	$.00013067
Vax 11/750	$36500.00	$.00033974

Table 7: Cost per second of using resources

Operation	Network Time	Processor Time	Disk Time
Message	1	18.49	0
Page Fetch	0	5.25	26.02
Tuple Processing	0	2.3	0

Table 8: Time (in milliseconds) per database operation on each resource

To compute the dollar based ratio of C to D to P, we must weight the measurements reported in Table 8 by these dollar amounts. The results are that a network message costs \$.006299 per site in the net, a page fetch costs \$.005313, and processing a tuple costs \$.0007814. This implies that the ratio of C to D to P is 8.0 to 6.8 to 1.

References

[Apers 83]P.M.G. Apers, A.R. Hevner, and S.B. Yao, "Optimization Algorithms for Distributed Queries," IEEE Transactions on Software Engineering, Vol. SE-9, No. 1, January 1983.

[Astrahan 76]M.M. Astrahan et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, June 1976.

[Bernstein 81a]P.A. Bernstein and D.W. Chiu, "Using Semi-Joins to Solve Relational Queries," Journal of the ACM, Vol. 28, No. 1, January 1981.

[Bernstein 81b]P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981.

[Chamberlin 76]D.D. Chamberlin et al., "SEQUEL: A Unified Approach to Data Definition, Manipulation, and Control," IBM J. Res. Vol. 20, No.6, 1976.

[Chamberlin 81]D.D. Chamberlin et al., "Support for Repetitive Transactions and Ad-Hoc Queries in System R," ACM Transactions on Database Systems, March 1982.

[Chu 69]W.W. Chu, "Optimal File Allocation in a Multiple Computer System," IEEE Transactions on Computers, Vol. C-18, No. 10, October1969.

[Chu 79]W.W. Chu, and P.Hurley, "A Model for Optimal Processing for Distributed Databases," Proc. 18th IEEE Compcon, Spring 1979, pp. 116-122.

[Chu 79]W.W. Chu, and P.Hurley, "Optimal Processing for Distributed Databases," IEEE Transactions on Computers. Vol. C-31. No. 9, September 1982.

[Codd 70]E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[Daniels 82]D. Daniels, and P. Ng "Distributed Query Compilation and Processing in R*," Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering, Vol. 5, No. 3, September 1982.

[Date 81]C.J. Date, "An Introduction to Database Systems," Third Edition, Addison-Wesley, 1981.

[Epstein 78]R.Epstein, M.R.Stonebraker and E. Wong, "Distributed Query Processing in a Relational Data Base System," Proc. ACM SIGMOD, May 1978, pp. 169-180.

[Epstein 80]R.Epstein and M.R.Stonebraker, "Analysis of Distributed Data Base Processing Strategies," Proceedings International Conference on Very Large Data Bases, Montreal, 1980.

[Epstein 80b]R.S.Epstein, "Query Processing Techniques for Distributed, Relational Data Base Systems," Ph.D. Thesis, College of Engineering, University of California, Berkeley, Memorandum No. UCB/ERL M80/9, 15 March, 1980.

[Fujitsu 79] "M2351A/AF Customer Engineering Manual," Fujitsu Corporation, Tokyo, Japan, 1979.

[Goldberg 83]A. Goldberg, S. Lavenberg, G. Popek, "A Validated Distributed System Performance Model," Proc. Performance 83, 1983.

[Goldberg 82]A. Goldberg, G. Popek, "Measurements of a Distributed Operating System: Locus," Unpublished paper, 1982.

[Goodman 79]N. Goodman et al., "Query Processing in SDD-1: A System for Distributed Databases," Computer Corporation of America, Tech. Rep. CCA-79-06, 1979.

[Haas 82]L.M. Haas, P.G. Selinger, E. Bertino, D. Daniels, B. Lindsey, G. Lohman, Y. Masunaga, C. Mohan, P. Ng, P. Wilms, R. Yost, "R*: A Research Project on Distributed Relational DBMS," IBM Research Report RJ 3653, IBM Research Laboratory, San Jose, CA., October 21 1982.

[Held 75]G.D. Held, M.R. Stonebraker, E. Wong, "INGRES - A Relational Data Base System," Proc. NCC, Vol. 44, 1975.

[Hevner 79]A.R. Hevner, and S.B. Yao, "Query Processing in Distributed Database Systems," IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979.

[Kirby]O.T.Kirby, "Architecture of LOCUS Pipes," LOCUS Group Memo 10, UCLA Department of Computer Science, School of Engineering and Applied Science, University of California, Los Angeles, December 3, 1982.

[McKusick 82]M.K.McKusick, W.N. Joy, S.J. Leffler, R.S.Fabry, "A Fast File System for UNIX," Draft of September 6, 1982, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1982.

[Metcalf 76] R.M. Metcalf and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, Vol. 19, No. 7, July 1976.

[Parker 81] D.S. Parker, Jr., G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, C. Kline, "Detection of Mutual Inconsistency in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 240-246.

[Popek 81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," Proceedings of the Eighth Symposium of Operating Systems Prinicples, Pacific Grove, CA, December 1981.

[Reiner 82] D. Reiner, A. Rosenthal,"Strategy Spaces and Abstract Target Machines for Query Optimization," Bulletin of the IEEE Technical Committee on Database Engineering, Vol. 5 No. 3 September 1982.

[Rosenthal 82] A. Rosenthal, D. Reiner, "An Architecture for Query Optimization," ACM SIGMOD Conf., Orlando, Florida, June 1982.

[Rothnie 80] J.B. Rothnie, P.A. Bernstein, S.A. Fox, N. Goodman, M.M.Hammer, T.A. Landers, C.L. Reeve, D.W. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)," ACM Trans. on Database Systems, 5,1 March 1980, pp. 1-17.

[Sacco 81] G.M. Sacco and S.B. Yao, "Query Optimization in Distributed Data Base Systems," Department of Computer Science, University of Maryland technical report 1981.

[Selinger 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," ACM SIGMOD 1979.

[Selinger 80] P.G. Selinger, and M. Adiba, "Access Path Selection in a Relational Database Management System," ACM SIGMOD 1979.

[Stonebraker 75a] M.R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," University of California Berkeley, Electronics Research Laboratory, Memorandum ERL-M514, March 1975.

[Stonebraker 75b] M.R. Stonebraker, E. Wong, P. Kreps, G. Held, "Design and Implementation of Ingres," University of California Berkeley, Electronics Research Laboratory, Ingres Distribution Tapes, December 1975.

[Stonebraker 77] M.R. Stonebraker and E. Neuhold, "A Distributed Database Version of Ingres," 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977.

[Stonebraker 83a]M.R. Stonebraker, J. Woodfill, J. Ranstrom, J. Kalash, K. Arnold, and E. Anderson, "Performance Analysis of Distributed Data Base Systems," University of California, Berkeley Memorandum No. UCB/ERL 83/12, 25 July 1983.

[Stonebraker 83b]M.R. Stonebraker, Personal communications, November 10, 1983.

[Theil 82]G. Theil, "Distributed Database and Distributed Operating System Design," Working Draft, August 1982.

[Theil 83]G. Theil, "Partitioned Operation and Distributed Data Base Management System Catalogs," Ph.D. Dissertation, University of California Los Angeles, 1983.

[Walker 83]B. Walker, "Issues of Network Transparency and File Replication in the Distributed Filesystem Component of LOCUS," Ph.D. Dissertation, University of California Los Angeles, 1983.

[Wong 76]E. Wong and K. Yousefi, "Decomposition- A Strategy for Query Processing," ASM Trans. on Database Systems, Vol. 1, No. 3 Sept. 1976.

[Wong 77]E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases," Berkeley Workshop on Distributed Data Management and Computer Networks, 1977.

[Yao 79]S.B. Yao, "Optimization of Query Evaluation Algorithms," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979.

[Zloof 77]M.M. Zloof, "Query-by-Example: A Data Base Language," IBM Systems Journal, Vol.16, no.4 1977.