QUERY PROCESSING STRATEGIES FOR DISTRIBUTED DATABASE SYSTEMS

Csaba Egyhazy

Department of Computer Science
Virginia Polytechnic Institute and State University
Falls Church, VA 20042

CS830016

Query Processing Strategies for Distributed Database Systems

1.0 INTRODUCTION

The identification of strategies and policies for optimal query processing using the relational data model [CODDE1] is of major concern to researchers in the field of distributed database systems. Simply stated, the problem is as follows: Once a user initiates a query, the system has to determine the best possible sequence and sites for performing the required relational operations.

Most previous models for studying query processing [BERNSP1], [WONGE1] [HEVNEA1] assumed a totally reliable network with sites containing non-duplicate and non-fragmented relations. The query processing strategy proposed by [ROTHNJ1] involved the transmission of all the necessary data to a control site for assembly and processing. With the use of a reducer, based on the semijoin operator concept, improvements in the performance of this strategy were achieved, refer to [BERNSP1]. The system defined in [HEVNEA1] is slightly less restrictive, in that it allows for possibly redundant portions of the database. In a sequel to it, [APERSP1] indicates that relation fragments may be considered as relations for distribution. The only allocation constraint is that all data must be either locally or globally accessible from any network node. Nevertheless, the subject of fragments was later dropped from further consideration.

Recently [YUC1] proposed a model where a relation may have a number of horizontal disjoint fragments or may have a number of copies, but not both.

2.0 DISTRIBUTED DATABASE MODEL

The model we propose consists of a distributed database system, allocated a priori across sites which may or may not be all interconnected. Furthermore, at any given time one or more sites may be down or are experiencing delays due to line contention. The values of these last two parameters of the model are generated dynamically by a Distributed System Simulator (DSS), described in detail in section 2.4.

Each site consists of a database, a DBMS and an Optimizer. The database is viewed logically in the relational data model. It may have horizontal disjoint fragments, copies of relations and/or unique relations. A relational DBMS is physically resident at each site. All incompatibilities affecting the flow of requests emanated from the initial query are assumed resolved. Thus, the distributed processing of a query is supported by compatible DBMSs. Once a query is received by the system the Optimizer determines the best strategy for processing it. This latter area is the focus of our current research efforts.

2.1 Model Implementation Environment

The environment that supports the implementation of the above described model consists of a VAX 11-780 under VMS, a special software product called Dialogue Management System (DMS) and a database management system called MDB. The last two products developed at Virginia Polytechnic Institute and State University.

Within DMS we employ the multiprocess execution environment, which allows the simultaneous execution of more than one process. A

process can be defined freely. In our particular case the following processes were established: Each host computer in the network is a process. The MDB corresponds to a distinct process, as is the Distributed System Simulator (DSS). Finally, the Optimizer, that determines the processing strategy for a query, is also defined as a process. Processes are individually activated (for instance for editing purposes), communication established between two or more processes and/or simultaneously executed. The latter accomplished by means of the DMS multiprocess execution environment facility. Figure 1 depicts the resource configuration for an emulation of a distributed database with four sites.

As seen from Figure 1 the Optimizer and DBMS, defined as processes 5 and 7 respectively, will communicate with each of the four sites or host computers. While the Distributed System Simulator (process 6) has a single connection, as it supplies traffic status information to the Optimizer at the time a query is received. The logical interface among processes is accomplished by means of the Dialogue Management System facility.

2.2 Dialogue Management System

A Dialogue Management System (DMS) is a special software facility being developed at Virginia Polytechnic Institute and State University for creating, modifying, and testing human-computer interfaces. DMS has been implemented [EHRICR1] using a variation of the rendezvous concept [USDOD1] in which two processes must synchronize before an interprocess dialogue can be initiated. In this implementation, an image called ENTRY is run under the user's login process.

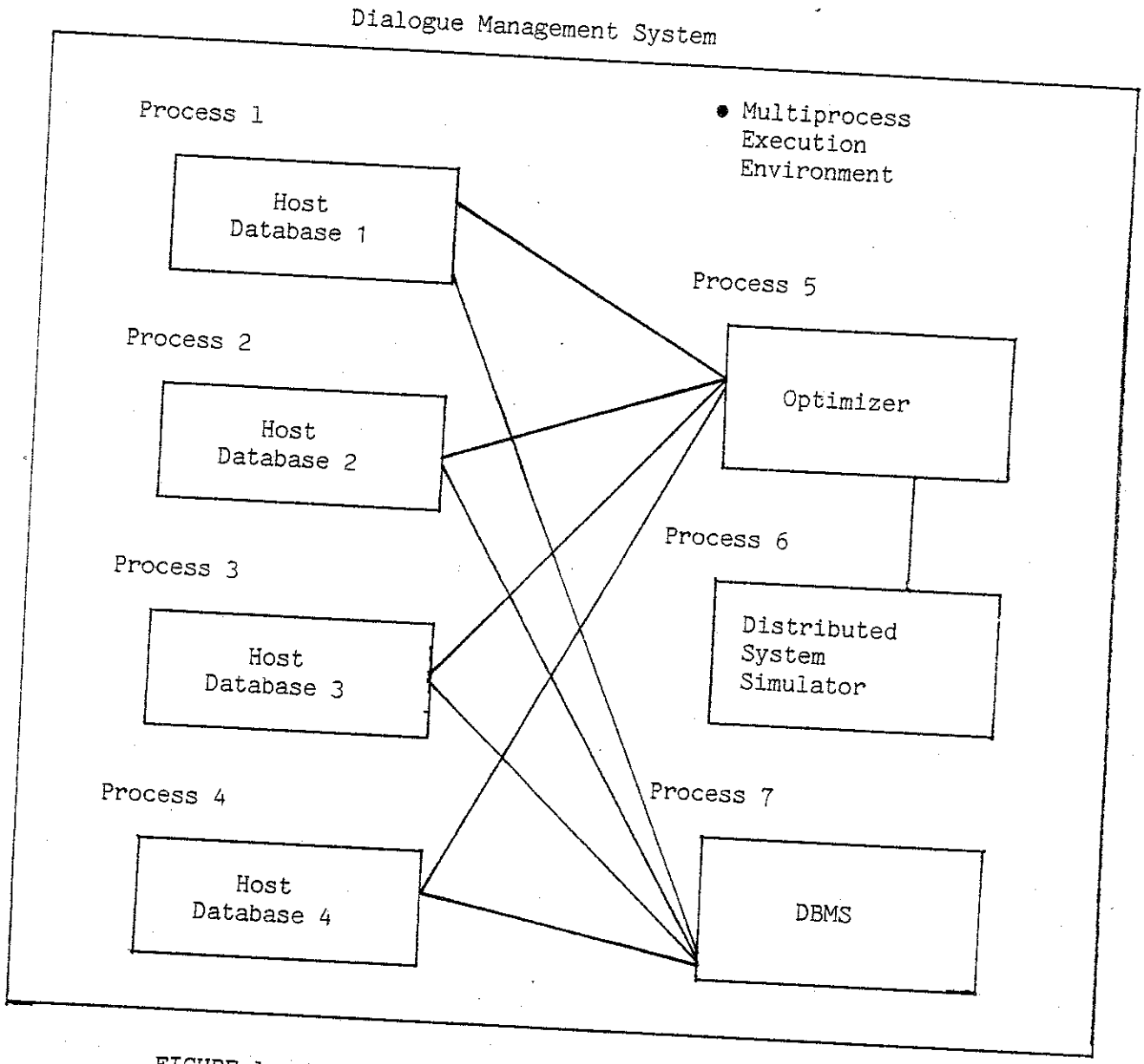


FIGURE 1 Four Site Distributed Databased System Emulation using DMS Multiprocess Execution Environment

ENTRY is the control executive of DMS, and all DMS processes are subprocesses of the process running ENTRY. Since it is a VAX/VMS implementation, the interprocess communication mechanism is called a mailbox, which is a portion of non-paged physical memory. Each process has one request mailbox whose name is based upon its process identification code, which is a unique number assigned by the operating system to the process when it is created. An image never writes to its own mailbox, which is treated as read only by its process. In each process except the one running ENTRY, mailboxes are serviced by software interrupts called a synchronous system traps (AST's). Thus, mailboxes are read almost instantaneously with one exception. When data (as opposed to control information) is being transferred AST's are disabled in the receiving process until the received data can be copied to its final destination to free the receive buffer.

Two processes are required to synchronize at the beginning of each interprocess dialogue. If process A requests a dialogue with process B, A sends B a request and hibernates until B gives A its attention. Then A and B can communicate freely, but with two constraints. In order to receive information from B, A must issue a collection request to B at some point before information is desired from B. This permits A to initiate concurrent dialogues with several processes and then order the processes from which it desires responses. The second constraint is that if B makes a third party request to C, it may only do so when A has finished transmitting data to B. Otherwise B would not be able to distinguish data

received from A from data received from C. With the exception of the preceding caution, A and B may freely exchange data in either direction as long as they choose.

As an executor received requests, they are queued internally and then served according to their priority. Once a request is served, the rendezvous is in effect until both processes decide to end the dialogue. With such an implementation the organization of an executor is quite simple, since it queues only requests and carries out only one interprocess dialogue at a time.

To complete the description of the DMS execution environment, a brief account of the communications that take place internally is given next. Suppose that image A is a computational program and that image B is a dialogue executor. When a user runs ENTRY, the user gives A's name in the default dialogue, and its process is created. When A executes a REQUEST, A creates a mailbox and informs ENTRY that it has done so. Since A has never communicated with B before, A asks ENTRY for the name of B's mailbox. ENTRY, however, has no knowledge of B either, so it creates a process that runs B. As soon as B executes an ACCEPT, B creates a mailbox and informs ENTRY. Next, ENTRY checks whether any image needed B's mailbox name, and it immediately sends the information to A, whose process is hibernating. A wakes up and now requests service from B and waits for acknowledgement, which comes immediately, since B's process is hibernating. Then data exchanges follow. If A later requests service from B again, ENTRY is not queried, since A remembers B's mailbox name. For a detail account of the DMS multiprocess execution environment refer to [EHRICR1].

2.3 Mini Database System (MDB)

MDB is a database management system being developed at Virginia Polytechnic Institute and State University, for relational databases. It is presently running on DEC VAX-11/780 with the VAX/VMS operating system. The system handles up to 41 relations, each relation may have from 1 to 36 attributes. The MDB provides a variety of commands which enable users to:

1. Define, load, store and drop relations,
2. Load and store files of tuples into and from a relations.
3. Insert, modify and delete individual tuples in a relation.
4. Query relations.
5. Rerun commands or use an editor to modify them.
6. Execute files of commands.

The MDB command language consist of 22 different commands. The user can enter commands one at a time interactively or can edit the commands into files which will be referenced and run in a program-like fashion. The Data Manipulation Language (DML) commands enables the user to transfer information in bulk to or from the database and to perform operations such as insert, modify, retrieve and delete individual tuples. Command Manipulation (CM) gives instructions to the MDB regarding the users terminal session and the use of Data Definition Lanuage (DDL) and DML commands. Additonally, VAX DCL commands can be entered while running the MDB program. This is an extremely useful feature, since the used does not need to leave the MDB environment to submit a DCL command.

2.4 Distributed System Simulator

The Distributed System Simulator (DSS) represents a software product that creates random data on the status of each host computer's local processing activities and the traffic across the network in general. A DSS is therefore a simulator that generates, among other things, data about queue length for both local processing and data transmission/reception for each host and among all connecting hosts respectively. The connectivity property is defined regardless of the means by which these host computers actually communicate. We can thus have hosts communicating through phone lines, dedicated high speed lines, satellite or any other existing mode. The main function of the DSS is to provide, at the time a query is received, a snapshot of the distributed system's operating condition, resembling as closely as possible the operational characteristics of most existing systems.

The Optimizer uses the information it receives from the DSS to identify only those query processing strategies that are both feasible and timely. Feasible in the sense that indeed the host is operational, both for local processing and/or data communications. Timely meaning that the estimated time for processing the query in a particular way satisfies the user's response time requirement. Therefore, if a query processing strategy is feasible, but involves excessive delays in local processing and/or data transmission, it is eliminated from further consideration. In the event of a deadlock, namely none of the identified strategies are timely, the user is prompted for a decision on either to increase the acceptable level

of the response time or to resubmit the query at a later time. The system then reacts accordingly.

3.0 OPTIMIZER: A CONCEPT

The Optimizer is hereby defined as the front end to the distributed relational database system, functionally responsible for determining the overall sequence of relational operations and defining the combination of local (those that refer to a particular site) operations that is both feasible and minimizes cost.

In order to build the conceptual framework that will facilitate achieving the above stated objective we need to define means of representing how a query flows through the system and define, within this flow, the role of the Optimizer. A query is expressed in relational terms using operations such as, selection, projection, join and semijoin. The relationship between these operators is not unique, since the same result can be obtained by applying the operators in a slightly different order [CHUW1]. Since these relationships can be depicted by graphs called query trees [HEVNEA1] and given the permutability properties of relational operators [EGYHAC1], in most non trivial cases there will be more than one query tree that produces the correct answer. A sample set, consisting of three permutability properties of semijoins, is given in Appendix A. Examining all the permutability properties of relational operators take up time and storage, and is therefore undesirable. The first major task of the optimizer is therefore to reduce the total number of possible feasible query trees. Among the most obvious heuristic rules in distributed query tree optimization we have:

(i) Duplicates, unless otherwise specified, have to be eliminated at every level of the query tree, since duplicates increase both processing and communication costs.

(ii) A sequence of selections or projections can be combined to form a single selection or projection.

(iii) In certain cases, we can combine a cartesian product and an adjacent selection to make a join. Similarly, joins and adjacent projections can be combined to form semi-joins.

(iv) Performing selections at the bottom level of a query tree eliminates tuples that are not necessary at the lowest level.

(v) In an unary operation, output volume of data is always less than the input volume of data. Hence, the placement of unary operators at the lowest level possible will reduce the volume of data flow from bottom to top.

(vi) If the result of a common subexpression is not a large relation, then it can be read from the secondary memory in much less time than it takes to compute it.

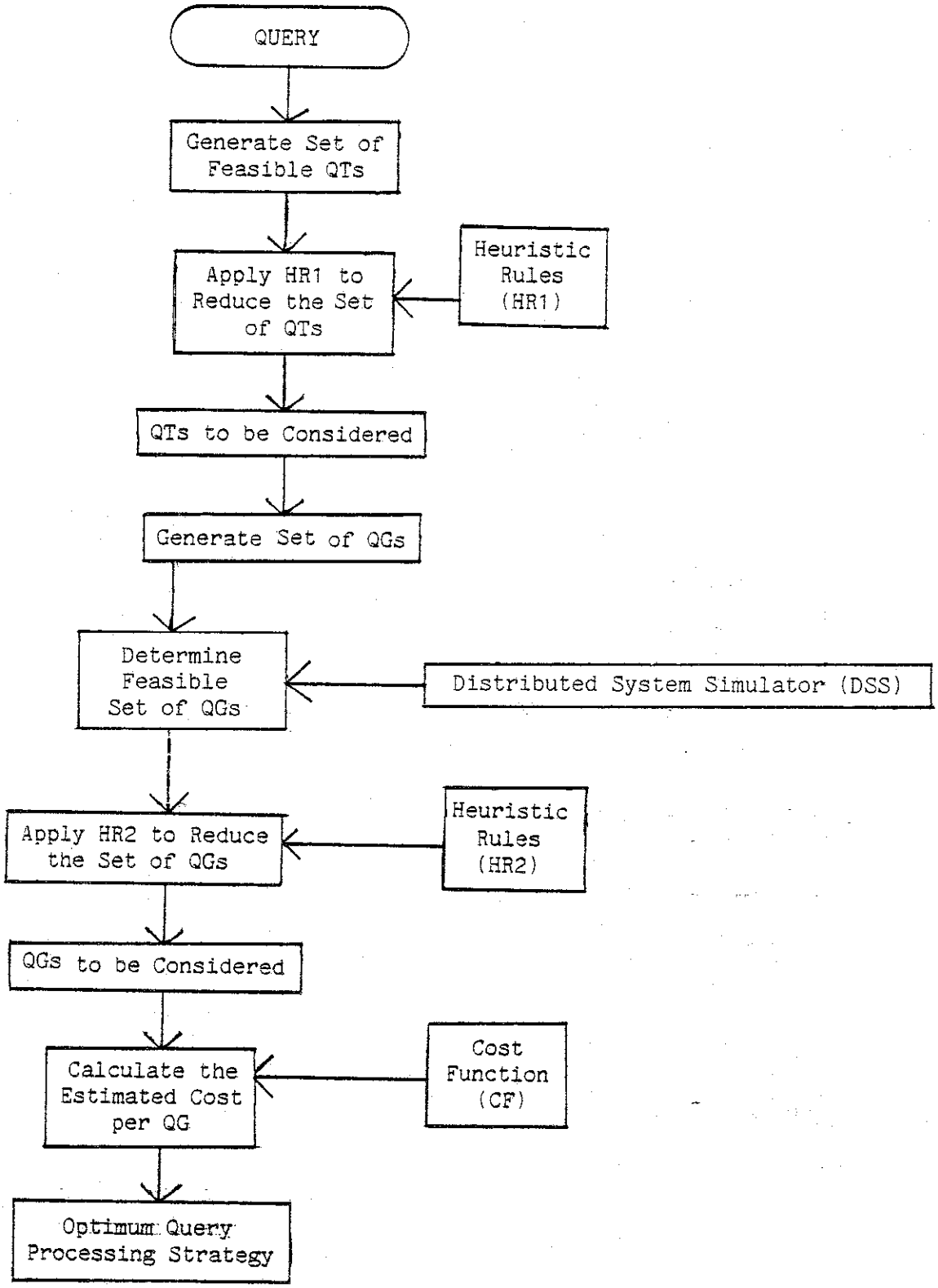
Other heuristic rules to support this activity are reported in [CHUW1] and [EGYHAC1]. The Optimizer merely automates their implementation and determines the reduced set of feasible query trees to be considered further. Since in distributed database systems it is often desirable to perform a group of operations at a single site and then transmit the results to other site(s) for further processing, we collect sets of connected operations in the query tree into groups called local operation groups [CHUW1]. These arrangements are represented by query processing graphs, which consist of execution nodes, storage nodes and arcs. The execution node represents

execution of a local operation group performed at a single site. The storage node represents a file stored at a specific site. Arcs connecting the execution nodes represent data communication between sites and arcs connecting storage nodes to execution nodes represent input data to the execution nodes. Therefore, a query processing graph not only represents a sequence of operations for processing the query, but also provides the information about a single operation or groups of operations to be performed at the same site or at different sites.

Since there are many possible ways of selecting local operation groups, there are many possible query graphs for a given query tree. Some of these are discarded by the optimizer as infeasible, based on information supplied by the Distributed System Simulator (refer to Section 2.4). For instance, there might be an excessive delay in either sending/receiving data between any two sites and/or processing an operation locally at a particular site. Therefore, the actual state of the network, given by the DSS, determines which are the feasible query processing graphs and which should not be considered any further unless the user is prepared to resubmit the query at a later time. At this point in the flow of the query the optimizer implements a second set of heuristic rules, such as those defined by [EGYHAC2], reducing the original set one more time. Some properties of query processing graphs useful in the reduction process are discussed further by [CHUW1].

In summary, at this point in the analysis of possible strategies to process the query we have established a feasible and reduced

set of query processing graphs. Among them, we are to select the one that minimizes a given cost function. Such a cost function can be found in [CHUW1]. The above described flow is shown pictorially in Figure 2.



QT = Query Tree
QG = Query Graph

FIGURE 2 FLOW DIAGRAM OF THE OPTIMIZER

4.0 IMPLEMENTATION

The theoretical constructs described above are in an early stage of implementation, particularly as it refers to the Distributed System Simulator and the Optimizer. The other components of the system, namely the Dialogue Management System Facility, the MDB database management system and the physical distributed database, are fully operational and are under going tests to ascertain their connectivity potential within the DMS multiprocess execution environment.

References

Query Processing Optimization in Distributed Systems

[KERSCL1] Kerschberg, L., Ting, P. and Yao, B. "Query Optimization in Star Computer Networks" ACM TODS Vol. 7, No. 4, December 82.

[BERNSP1] Bernstein, P. A., Goodman, N., Wong, E., Reeve, C. and Rothnie, J., "Query Processing in a System for Distributed Databases" ACM TODS, Vol. 6, No. 4, December 1981.

[SMITHJ1] Smith, J. M. and Change, P. Y. "Optimizing the Performance of a Relational Algebra Database Interface" Communications of the ACM, Vol. 18, No. 10, October, 1975.

[HEVNEA1] Hevner, A. R. and Yao, S. B. "Query Processing in Distributed Database Systems" IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May,. 1979.

[CHUW1] Chu, W. W. and Hurley, P. "Optimal Query Processing for Distributed Database Systems" IEEE Transactions on Computers, Vol. C-31, No. 9, September, 1982.

[WONGE1] Wong, E. "Retrieving dispersed data from SDD-1" Prac. Berkeley Workshop Distributed Data Management and Computer Networks, May, 1977.

[APERSP1] Apers, P., Hevner, A. and Yao, B. "Optimization Algorithms for Distributed Queries" IEEE Transactions on Software Engineering, Vol. SE-9, No. 1, January, 1983.

[USDOD1] United States Department of Defense "Reference Manual For the ADA Programming Language," DARPA, July, 1980.

[EHRICR1] Ehrich, R. W. "The DMS Multiprocess Execution Environment" Department of Computer Science, Virginia Polytechnic Institute & State University, June 2, 1982.

APPENDIX A

I. PERMUTABILITY PROPERTIES OF SEMIJOINS

The permutability properties of semijoins with other unary and binary operations are developed by combining unary operators (selection, projection) and binary operators (difference, division, union, intersection, join and semijoin) in a particular order.

The notation to be used is as follows:

Relations: A, B and C

PROJECTION (π): projects its operand into the set of attributes s.

SELECTION (σ_p): selection condition p.

DIFFERENCE (-): left hand side relation minus right hand side relation

DIVISION ($\div$): left hand side relation divided by the right hand side relation

UNION ($\cup$)

INTERSECTION($\cap$)

NATURAL JOIN ($A \bowtie B$): applicable only when both A and B have columns that are named by attributes.

SINGLE DOMAIN SEMIJOIN $(A \ltimes B) = \pi_A(A \bowtie B)$

$(A \rtimes B) = \pi_B(A \bowtie B)$

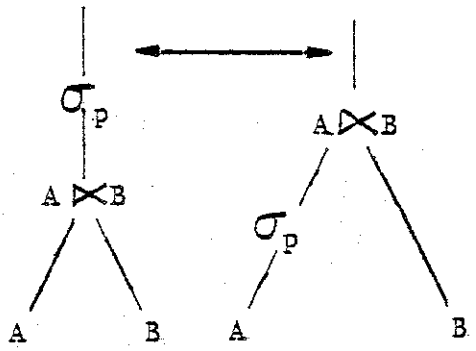
1. Semijoin with Adjacent Unary Operations

Let us consider the selection and projection operations

(i) Semijoin with adjacent selection: When a selection is followed by a semijoin we can exchange them without modifying the resultant query. If a semijoin on A, i.e., $(A \ltimes B)$, follows a selection condition p an exchange is feasible only if all the attributes in condition p are attri-

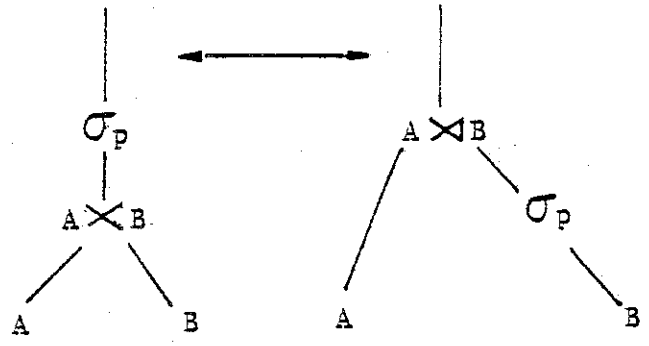
butes of relation A. While, if a semijoin on B, i.e., (A⋉ B), follows a selection condition p an exchange is feasible only if all the attributes in condition p are attributes of relation B. Both of these bidirectional transformations are illustrated in Figure I.

(ii) Semijoin with adjacent projection: When a projection is followed by a semijoin we can exchange them without modifying the resultant query. The bidirectional transformation shown in FigureII can be performed only if all the attributes deleted in the projection do not intersect with the semijoin's selection attributes.



The left to right transformation above can be performed only if all the attributes in condition p are attributes of relation A.

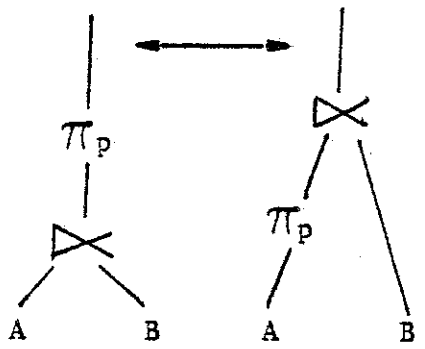
(a)



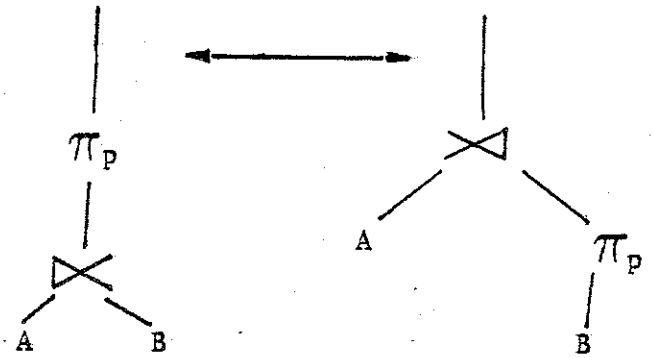
The left to right transformation above can be performed only if all the attributes in condition p are attributes of relation B.

(b)

FIGURE I Semijoin with adjacent selection



(a)



(b)

The left to right transformations can be performed only if all the attributes deleted in the projection do not intersect with the semijoin's selection attributes.

FIGURE II Semijoins with adjacent projection

2. Two Identical Binary Operations in Sequence

(i) Two semijoin operations in sequence: Since semijoin operations are commutative and associative, if we project on the same set of attributes on both operations we can exchange the relations that project back after the join operation. In Figure III they correspond to relations B and C. The bidirectional transformation holds for both equality and inequality semijoins.

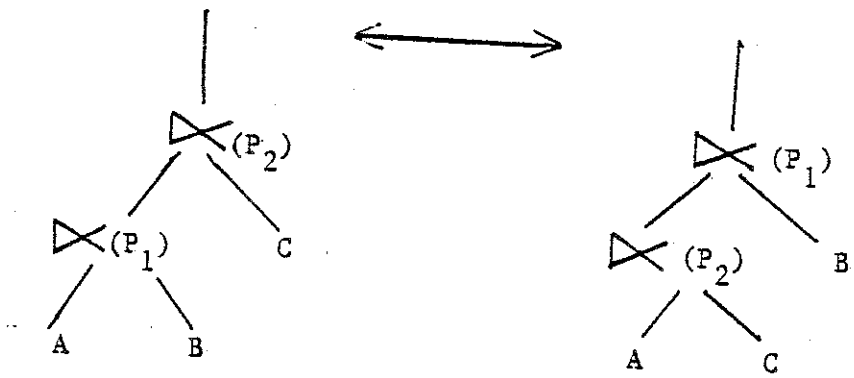


FIGURE III Semijoin operations in sequence

3. Semijoin operations with other binary operators

(i) Semijoin and union: Semijoin can not always be permuted with a subsequent union operation, i.e.:

$$(A \ltimes B) \cup C \neq (A \cup B) \ltimes (A \cup C)$$

If a union operation is operated before a semijoin, the sequence of operations results in a larger set than if we perform the semijoin. The reason is that large files are produced after a union operation which when accompanied by a semijoin gives a large set of tuples. The sequence of operations mentioned above may be equal in some cases but it is not universal.

4. Semijoin and Join

Semijoin can be permuted with join if we apply the same condition p. Thus,

$$(A \ltimes B) \bowtie C = (A \ltimes C) \ltimes B$$

Figure IV depicts the above property.

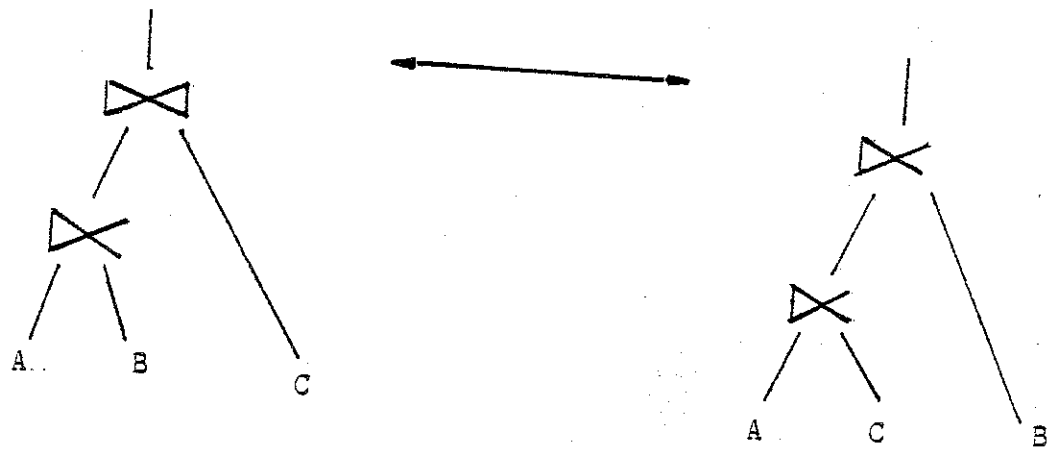


FIGURE IV Semijoin and Join