# A Parallel Processing Strategy for Evaluating Recursive Queries

Louiqa Raschid
Stanley Y.W. Su

Database Systems Research and Development Center

University of Florida

## Abstract

The set of resolvents generated by a recursive intension in a first-order database is treated as a set of concurrent database queries. A strategy for efficiently evaluating these concurrent queries in a multi-processor environment is presented. The strategy combines three query processing techniques, namely, query decomposition, intermediate result sharing and data-flow and pipelined query execution to achieve a high degree of parallelism. An analytical study uses the response time for each resolvent and the execution time for a set of resolvents as a performance measure to examine the performance gain due to the data-flow and pipelined approach to query processing.

## 1. Introduction

Knowledge base management systems (KBMS) is a new technology resulting from the integration of techniques from database management systems (DBMS) and artificial intelligence (AI) [BRO84, GAL83, JAR84, KEL82, KER84, RAS85, SU85, WIE84]. A KBMS requires a powerful language for defining various kinds of knowledge rules including integrity and security constraints, deductive rules to generate new information, rules for describing properties such as transitivity, symmetry, etc., as well as domain specific expert rules. To support this, query languages have to be enriched; one such extension is to permit the use of general logical clauses in the query languages of relational databases. As a result of such an extension, queries may be defined recursively and straightforward methods of query evaluation may fail. A variety of strategies have been proposed to deal with recursive queries [HAN86, NAQ84 and ULL85], and in section 2 we examine schemes that generate a series of resolvents that provide answers to a recursive query.

Efficiency is a key factor in the succesful integration of AI and DBMS technology. In this paper, we present techniques for the efficient implementation of recursively defined queries in KBMS. More specifically, we treat the sequence of resolvents generated by a recursive clause as a set of concurrent database queries and apply query processing techniques to optimize the evaluation of these concurrent queries.

In section 3, we describe a strategy which combines three known techniques, namely query decomposition, intermediate result sharing, and pipelining and data-flow based approaches to query execution. In section 4, we apply this strategy in the evaluation of a simple recursive rule that defines the transitive closure of a database relation. We identify operations that are candidates for parallel evaluation as well as operations that can benefit from result sharing. We then study the effect of pipelined execution on this evaluation strategy. In section 5, we study the effect of pipelining on this evaluation strategy, using the execution time for evaluating the set of resolvents and the response time of each individual resolvent as performance measures. In section 6, we discuss relevant extensions to this work that involve general recursive clauses.

## 2. Methods for Evaluating Recursive Queries

It is assumed that the reader is familiar with the relationship between logic programming and relational databases [BRO84, GAL83, JAR84, REI78a, REI78b] and the resolution principle in theorem proving [ROB65]. A first-order database is a function-free first-order theory in which the extensional database (EDB), corresponding to the data in relations, is a set of ground (having no variables) positive unit clauses. If we consider a Horn database, then the intensional database (IDB) is a set of Horn definite clauses with exactly one positive literal. Each clause of the IDB represents a definition of some of the tuples named in its positive literal, which could also be an EDB predicate. For example,

P(x,z) :- Q(x,y), R(y,z)

says that the appropriate join (over y) of Q and R is contained in P. The set of tuples in P is the union of all tuples provided by each intensional clause defining P as well as all EDB tuples if P is an EDB predicate as well.

Straightforward methods for query evaluation are insufficient in the presence of recursive definitions. Recent research has focused on this problem [CHA81, HAN86, MIN81, NAQ84 and ULL85]. In [ULL85], methods for implementing queries that are expressed using first order logic as a collection of Horn clauses are reported. A rule/goal tree is built using the rules (Horn clauses) and goals (terms). A rule/goal tree is equivalent to an expression in relational algebra and, for a finite tree, a bottom up evaluation will build a relation at each node until the root is evaluated. Recursive rules result in potentially infinite rule/goal trees. The paper presents a limit of trees process to evaluate infinite trees.



**Proceedings of the Twelfth International Conference on Very Large Data Bases**

**Kyoto, August, 1986**

Significant contributions of [ULL85] include the use of capture rules which specify under what circumstances a node (of a rule/goal graph constructed using the logical rules) can be evaluated and provide an efficient implementation strategy for evaluating these trees. Two methods to terminate rules that involve recursion are given; one takes advantage of the finiteness of the domains and this is the method we adopt.

In [NAQ84], the problem of deriving a set of database retrieval requests, which gives the correct answers to a query involving a recursive statement and is guaranteed to terminate, is addressed. In this work, the clauses of the IDB are represented as a connection graph (CG) [SIC76]. A recursive intension occurs in a CG as a special form of cycle called a potential recursive loop (PRL). Only PRLs lead to database retrievals containing recursive statements and algorithms for detecting PRLs are well known [SIC76].

Consider an example database relation such as Edge(start_node, end_node) which has a tuple for each direct edge between two nodes in a graph. Then, the transitive closure of the relation Edge would be a relation, Reach, which has a tuple for any two nodes in the graph that has a path between them. The following Horn definite clauses will define the transitive closure, Reach, of the Edge relation:

Reach(x1,y1) :- Edge(x1,y1) (1)
Reach(x1,z1) :- Reach(x1,y1), Edge(y1,z1) (2)

Using the CG and resolving around the PRL, a query such as Reach(?,c), where c is a constant, i.e., a query that retrieves all nodes that have a path to node c, will generate the following resolvents:

Edge(?,c) (3)
Edge(?,y1), Edge(y1,c) (4)
Edge(?,y2), Edge(y2,y1), Edge(y1,c) (5) etc.,

A general algorithm for retrieving answers from the database, based on these resolvents is presented in [NAQ84]. The algorithm consists of an outer loop (corresponding to each resolvent) and two inner loops. Initially, using selection on the database relation Edge, values for y1 will be pushed on a queue. Then, for each resolvent, all answers will be extracted in the first inner loop (using appropriate join, selection and projection operations) and the corresponding set of values for the next resolvent, eg., the values for y2, will be queued in the second inner loop.

The time for serially evaluating several resolvents, on a single processor system, will be very long. What we have studied is a strategy for the parallel evaluation of these resolvents which are generated by the recursive intensions, on a multi-processor system. Parallel evaluation of the resolvents will eliminate the outer loop. In addition, identifying common subexpressions in these resolvents will allow intermediate result sharing among these parallel operations, thus simplifying the operations in the inner loops and allowing the two inner loops to be executed simultaneously. Further, the evaluation strategy gains additional parallelism by using a pipelining approach for executing database operations.

In [HAN86], the performance of several algorithms for processing a recursive query are compared. The process of applying a recursive rule and generating longer resolvents is compared to a wavefront i.e., the saved result of an operation is used to derive a new result. The algorithm DW (or double wavefront) is similar to our strategy (descibed in section 4) in the manner it shares results among resolvents. However, they do not treat the resolvents as a set of concurrent queries nor do they consider horizontal and vertical concurrency and pipelining techniques to increase the degree of parallelism and to improve the efficiency of execution.

## 3. The Impact of Query Processing Techniques

In the previous section, a query of the form Reach(?,c) generated a sequence of resolvents that had to be evaluated to provide answers to the query. Each of these resolvents can be considered a query against the database and the set of resolvents can be treated as a set of concurrent queries; query processing and optimization techniques can then be used to optimize the execution of these concurrent queries.

Query decomposition is a process of translating a query into a hierarchy of primitive operations; the result is a query tree in which the nodes represent the primitive operations [AST76, ROT80, STO76, WON76]. The advantage of query decomposition is that it identifies primitive operations on different branches of a query tree that can be executed in parallel (i.e., "horizontal" concurrency), thus increasing the degree of parallelism. It also increases the probability of finding an overlap among several query trees which facilitates intermediate result sharing.

The sharing of intermediate results among concurrent queries and the resulting elimination of redundant execution of operations have been proposed in [FIN82 and JAR84]. In [BOR84, CHO85], it has been shown that as the degree of sharing among concurrent queries increases, the query throughput also increases. Most of this research studies the effect of eliminating low-level read operations by sharing buffer space. More recent work [MIK85, SU86], shows the advantage of sharing the output of high-level operations such as select, join, etc.

Both query decomposition and intermediate result sharing have an impact on the evaluation of the concurrent resolvents. Each resolvent is equivalent to a relational algebra expression [ULL85]; thus, the set of concurrent queries or resolvents can be decomposed into a hierarchy of primitive algebraic operations some of which can be evaluated in parallel. The feature of a recursive intension is that each time the recursive clause is applied it generates a longer resolvent which is an extension of a previous resolvent [NAQ85]. Thus, there is a potential for identifying common sub-expressions and sharing intermediate results of high-level algebraic operations among the concurrent queries. For example, on examining the resolvents in expressions (3), (4) and (5), we see that (3) is a sub-expression of (4), (4) is a sub-expression of (5), etc.

The third technique is the pipelining and data-flow based processing approach proposed for several database machines [BIC81, BOR80, FIS84, HON84, KIM84]. Using this technique, each processor assigned to a node in a query tree transmits a block of information as soon as it is produced. This is in contrast to traditional distributed systems that delay output until the operation assigned to the node is completely executed. The main advantage of this data-flow based approach is the possibility of "vertical" concurrency, where an operation at one level that requires input from an operation at a previous level can get its input data at an earlier instant, before the operation at the previous level is completed. Pipelining has a significant impact on the efficiency of concurrent queries that share intermediate results (such as the resolvents of a recursive intension), since the processors can get their shared data earlier, and start execution sooner.

## 4. Efficient Implementation of Recursive Queries

In this section, we describe a strategy which applies the query processing techniques of section 3 to process a set of concurrent queries (resolvents).

We first decompose each resolvent (query) into a hierarchy of primitive (algebraic) operations that can benefit from pipelining. We identify possible parallelism in executing these primitive operations as well as opportunities for intermediate result sharing among the resolvents (queries). Next, we study the effect of pipelining on this evaluation strategy. Finally, we determine a termination condition to halt the evaluation of these concurrent queries.

Although it is advantageous to maximize both the number of primitive operations being executed in parallel as well as intermediate result sharing among queries, the degree of parallelism is limited by the availability of processors and the amount of result sharing is limited by the bandwidth and the structure of the interconnection network. As the resolvents become longer (by the repeated application of the recursive rules), there is increasing opportunity for parallelism and there are several ways to decompose each resolvent into primitive operations. However, to maximize the opportunity for result sharing, it is desirable to decompose each resolvent so that it can share the greatest common sub-expression from a previously evaluated resolvent. In addition, to improve execution efficiency, the decomposition must first process operations on restricted relations, e.g., execute selection before join. To avoid irregularity in the interconnection network and to simplify the network structure, it is desirable to limit the sharing of results only between adjacent resolvents.

We use the example of the transitive closure, T, of a database relation A with two attributes of interest. T is defined as follows:

T(x,y) :- A(x,y)
T(x,z) :- T(x,y), A(y,z)

For convenience, we represent the database relation as A-ff where each "f" indicates an attribute that is free (unbound). We use "b" to indicate a variable bound to a constant value (i.e., a selection based on an attribute value). Thus, A-bf is the result of selecting tuples from the database relation A, based on the value of the first attribute.

Consider an example query which is a verification of the form T(a,c), where a and c are constants. This would correspond to finding all paths between two given points of a graph. Then, to answer this query, a series of database queries (resolvents) Ti-bb, as seen in Figure 4.1, will be generated, where "i" identifies the depth of the resolvent and "b" indicates each variable (or attribute of a relation) that is bound to a constant. T1-bb corresponds to the expression A(a,c) and the corresponding query will be (A-bb), which is a direct selection of tuples from the relation A. The second resolvent, T2-bb, corresponds to the expression A(a, y1), A(y1, c) where y1 is unbound and will be represented by

(A-bf JN A-fb)

which is identified as a primitive operation in our evaluation. This primitive operation comprises initial selections, A-bf and A-fb, from the relation A (corresponding to binding a variable in a predicate to a constant), a subsequent join (JN) operation over the appropriate attribute, here y1, (corresponding to variable binding between clauses) followed by a projection operation to produce answers corresponding to T2-bb of the form T(a,c). The primitive operation we have just described is typical of the operations resulting from the decomposition of resolvents. If the variables are not bound, then the initial selection will be omitted.

Resolvent T3-bb will be hierarchically decomposed into

( (A-bf JN A-ff) JN A-fb)

where (A-bf JN A-ff) will be evaluated first. Resolvent T4-bb will also be hierarchically decomposed into

( (A-bf JN A-ff) JN (A-ff JN A-fb) )

Figure 4.2 shows the hierarchical decomposition of the resolvents into primitive operations. Several of these operations can be executed in parallel. The legend [m]-j in the figure represents those primitive operations at level m that can be executed in parallel. The level, m, of the operation is different from the depth, i, of the resolvents and the value of m is determined by the input requirements. For example, operations at level 1, represented by [1]-j, are at the lowest level in the tree and do not require input from any other operation. However, operations at level 2, [2]-j, are those that require input from a previous level, in this case, from operations at level 1, etc. The value of j serves to distinguish between parallel operations at the same level.

Those expressions that are common sub-expressions are labelled in the figure by "common." For example, resolvents T3-bb and T4-bb have (A-bf JN A-ff) in common. To maximize result sharing, each resolvent is decomposed so that it can share the maximum common sub-expression from its immediate predecessor, i.e., the first resolvent is decomposed into its primitives and then the next resolvent is decomposed so as to share as many sub-expressions as possible from the previous resolvent, etc. For example, the resolvent T6-bb is decomposed into

[1]-3 [2]-3 [3]-2 [2]-4 [1]-4
( ( (A-bf JN A-ff) JN A-ff) JN (A-ff JN (A-ff JN A-fb) ) )

rather than an alternative decomposition of

[1]-3 [2]-p [1]-5 [3]-q [1]-4
( ( (A-bf JN A-ff) JN (A-ff JN A-ff) ) JN (A-ff JN A-fb) )

The first decomposition is based on using the largest sub-expressions, namely, the output of operations [2]-3 and [2]-4. The second decomposition does not maximize result sharing. It is also less efficient as it includes an operation [1]-5 which computes the join of two unrestricted relations.

Figure 4.3 illustrates the advantages of maximizing result sharing and maintaining a regular interconnection structure; i.e. it results in an evaluation strategy that is regular with respect to primitive operations and interconnections. This regularity is advantageous if special-purpose hardware is to be built to execute recursive queries. In this figure, the boxes at each level represents the primitive operations that can be executed in parallel at that level. The level, m, does not necessarily correspond to the depth, i, of the resolvent Ti being evaluated at that level. For example, at level 2, operation [2]-1 and [2]-2 produce output for resolvents T3-bb and T4-bb, respectively. All the operations do not produce answers to the query; some operations evaluate sub-expressions. For example, operations [2]-3 and [2]-4 evaluate the sub-expressions T3-bf and T3-fb, respectively.

Next, we examine the termination condition, based on the finiteness of domains. If the relation A is finite, then the transitive closure T is also finite. Referring to figure 4.3, if at any level, m, the operation [m]-3 did not produce any output,

i.e., $T_{m+1}$-bf was empty, then the processing can be terminated at that level m, since answers cannot be produced at subsequent levels (m+1). This also holds for the operation [m]-4 which evaluates $T_{m+1}$-fb. However, in the case of cyclic databases (databases that have cyclic relations, eg., {(a,b), (b,a)} ), determining the termination condition is more complex. Here, the operations [m]-3 and [m]-4 could produce output without necessarily producing new answers at subsequent levels. The system must check the output of the operations [m]-3 and [m]-4 and determine that there are tuples produced in $T_{m+1}$-bf and $T_{m+1}$-fb which will indeed produce answers, i.e., $T_{m+1}$-bf must be compared with the set {A-bf, T2-bf, .... Tm-bf} to ensure that $T_{m+1}$-bf is not a sub-set of this set. If $T_{m+1}$-bf is indeed a subset of this set, then no new answers will be produced at subsequent levels. Execution should terminate at that level. The same holds for $T_{m+1}$-fb.

## 5. Performance Evaluation

In this section, we evaluate the performance of the transitive closure algorithm. In our evaluation, we compare the performance of this algorithm with and without pipelining.

We compare the "distributed" approach which uses only horizontal concurrency, with the data-flow and pipelining based approach which uses both horizontal and vertical concurrency [MIK85, SU86a]. In both cases, we assume that parallel execution of primitive operations by multiple processors and intermediate result sharing are exploited. With the pipelined approach, a block of data is transmitted as soon as it is produced. A block is the operand granularity for input and output (of results). Processing at level (m+1) commences as soon as operations at this level have a block of data at their input nodes. This results in vertical concurrency across several levels. With respect to recursive intensions, this implies that several resolvents Ti-bb, will be evaluated in parallel.

The two performance measures used in this study are the response time (or the time to produce the first block of data) and the execution time (or the time to complete processing an operation). We measure the response time of each resolvent or query, Ti-bb, and the execution time for the set of concurrent queries, for some depth, i, of these resolvents. We expect that the pipelined approach will have better response time and execution time since each level will commence execution at a much earlier instant, as compared to the distributed approach.

For our evaluation, we use the simple hash join to model a primitive operation. The analysis of main memory resident database systems, in [DEW84], suggests that hash based query processing strategies are advantageous. The same result is reported in [MIK85, SU86a] for the data flow and pipelined approach. We assume that the hash tables fit into main memory.

Let the two relations to be joined be R1 and R2, and let their sizes (number of tuples) be k1*B and k2*B, respectively, where B is the block size expressed as the number of tuples in a block. Assume $k1 \geq k2$. Let Tbr be the time to input a block, Th the time for hashing the value of an attribute over which the join is to be performed, Tw the time to write a tuple in memory, and Tc the time to compare a hashed value with values in the stored hash table. Let j be the join selectivity defined as

j = (number of join tuples output) / k1*k2*B*B

Using values in [DEW84], we set values of 9, 20 and 3 microseconds for Th, Tw and Tc, respectively. The time for a sequential I/O operation was set at 10 milliseconds per page, for a page size of 40 tuples. In our analysis, the block size, B, is a parameter; thus, we vary the value of Tbr from 5 milliseconds (B = 20) to 25 milliseconds (B = 100). For blocks of shared results, the input time will be the transfer time across the network. We assume the same values for the transfer time as for the sequential I/O operation. We assume a selectivity factor, s, for both A-bf and A-fb of 10 percent. We do not vary s, as it only occurs at the first level and its effect is negligible.

For the distributed approach, the smaller relation, R2, will be read first, hashed and the hash table is stored in memory. The larger relation, R1, will then be read, hashed and compared with the stored hash table. Note that a 20 percent overhead accommodates the extra comparisons required when comparing values using a hash table [DEW84]. If there is a match, then the two matching tuples will be output. The selection only occurs at level 1 and will be considered as part of the input time. Any final projections will be included in the time to move the join output tuples to the buffer. The time spent to transmit the final result is also the input time of the next level operation(s) that use this result. However, we do not assume overlap in the I/O and processing times of an operation. The execution time of the primitive operation in the distributed case is the same as the response time, since output is not transmitted until processing is complete. It is the following:

time to read, hash and store tuples of R2 +
{ = Tbr*k2 + Th*k2*B + Tw*k2*B }
time to read, hash and compare tuples of R1 +
{ = Tbr*k1 + Th*k1*B +Tc*k1*B*1.2 }
time to output tuples of join result
{ = Tw*2*j*B*B*k1*k2 }

In the pipelined approach, the first block of R2 will be read, hashed and stored in memory. The first block of R1 will be read, hashed, compared with the current contents of the hash table and the join output, i.e., the pairs of matching tuples from both relations will be written into an output buffer. R1 will also be stored in the hash table for further comparison with subsequent blocks of R2. The subsequent blocks of R1 and R2 will be treated in a similar fashion. As soon as the number of tuples in the output buffer exceeds B, a block of output will be transmitted. After the last (k2-th) block of R2 is processed, the remaining blocks of R1 need not be stored in the hash table.

For each block i, where i = 1,...,k1, $Tin\text{-}R1_i$ and $Tin\text{-}R2_i$ is the time to read, hash and store (optional) blocks, respectively. $Tcomp_i$ is the time spent in comparing hashed values with the hash table and $Tout_i$ is the time spent in output of the join result. For i = 1 the following hold:

$Tin\text{-}R1_1$ =Tbr + Th*B + Tw*B;
$Tin\text{-}R2_1$ =Tbr + Th*B + Tw*B;
$Tcomp_1$ = Tc*B*1.2; $Tout_1$ = Tw*2*j*B*B

For subsequent blocks i = 2,3,..,k2, the following hold:

$Tin\text{-}R1_i = Tin\text{-}R1_1$; $Tin\text{-}R2_i = Tin\text{-}R2_1$
$Tcomp_i$ = Tc*2*B*1.2; $Tout_i$ = Tw*2*j*(2*i - 1)*B*B

For blocks i = k2+1,..,k1, the following hold:

$Tin\text{-}R1_i$ = Tbr + Th*B; $Tin\text{-}R2_i$ = 0
$Tcomp_1$ = Tc*B*1.2; $Tout_i$ = Tw*2*j*k2*B*B

In [MIK85, SU86a], the output rate of the pipelined join was averaged over the execution time to obtain an average rate of output. This is not very accurate since as more input blocks are accumulated, a single block of input will be compared against several blocks and the number of output tuples produced will increase. We have modeled a varying output rate for the join operation. The following discussion elaborates our model. In any data-flow based algorithm, the rate of output blocks produced by an operation is determined by the availability of input, as long as the i-th block of input can be completely processed before the (i+1)-th input block is available, i.e., the output rate is determined by the input rate (which is the output rate of the previous operation providing the input). At some point, the input blocks are available at a faster rate than they can be consumed. This is the critical point and after this point, the output rate is determined by the processing rate of the operation, itself.

Figure 4.3 shows that a sequence of operations, [1]-3, [2]-3, .. [m-1]-3, etc., controls the availability of input for [m]-1 and can be considered a critical path for [m]-1. In Figure 5.1, we have a sample graph describing the rate of producing output blocks as a function of the number of input blocks consumed, for the critical path operation [2]-3. The relationship between the number of input blocks consumed (bi) and the number of output blocks produced (bo) is the following:

j*bi*bi*B*B = bo*B if j*k2*k2*B*B > bo*B --- 5.1

j*bi*k2*B*B = bo*B otherwise --- 5.2

The time for operation P to consume (process) bi blocks of input is defined as Tproc(P,bi) and the time to produce bo blocks of output is defined as Tprod(P,bo). Figure 5.1 shows plots for two values of N (the number of tuples in the database relation A-ff) equal to 10000 and 200000 and different block sizes (B = 20 and B = 40). Our results show that the output rate increases gradually with increasing input being consumed. Assuming that the processing speeds of the operations along the pipeline are matched, the critical point for an operation, at level k, corresponds to the first block of input (bc) at level (k-1) that produces more than one block of output. The critical point is the least value of bc satisfying the following:

j*(2*bc - 1)*B*B > B

The response time and execution time for operations at level m will be defined recursively with respect to operations at level m-1 which provide input. For level 1 operations, these values can be obtained directly using the expressions for $Tcomp_i$, $Tout_i$, etc.

The response time for any primitive operation $P_i$, $Tres(P_i)$, will be a function of r1, the number of input blocks needed to produce the first output block. By substituting (bo=1) in either 5.1 or 5.2, the value of r1 (=bi) can be obtained. $Tres(P_i)$ is also a function of the response time of the operation(s) providing input, $P_{i-1}$ and $P_{i-1}'$ . Operation $P_i$ requires r1 blocks of input, each, from $P_{i-1}$ and $P_{i-1}'$ , to produce the first block of output. In all our experiments, the r1 blocks of input were produced before the critical point; thus, the response time is also determined by the output rates of the operations $P_{i-1}$ and $P_{i-1}'$ . The following holds:

$Tres(P_i)$ = maximum of the response times of $P_{i-1}$, $P_{i-1}'$
{ max [ $Tres(P_{i-1})$, $Tres(P_{i-1}')$ ] }
+ maximum time for $P_{i-1}$, $P_{i-1}'$ to produce r1 blocks
{ max [ $Tprod(P_{i-1},r1)$, $Tprod(P_{i-1}',r1)$ ] }
Note that this time must be adjusted to account for the fact that the first block is already available.

To determine the execution time of operation Pi, $Texec(P_i)$, we first determine the critical point of operations, $P_{i-1}$ and $P_{i-1}'$ , which provide input, and the corresponding output block, pc or pc', produced at the critical point. Before the critical point the output rate will be controlled by $P_{i-1}$ (or $P_{i-1}'$ ), and after the critical point the output rate will be controlled by operation $P_i$. If operation Pi processes (consumes) a maximum of k1 blocks, then the following holds:

$Texec(P_i)$ = maximum of the response times of $P_{i-1}$, $P_{i-1}'$
{ max [ $Tres(P_{i-1})$, $Tres(P_{i-1}')$ ] }
+ max. time for $P_{i-1}$, $P_{i-1}'$ to produce pc, pc' blocks, resp.
{ max [ $Tprod(P_{i-1},pc)$, $Tprod(P_{i-1}',pc')$ ] }
+ time for $P_i$ to process a max. of (k1-pc) or (k1-pc') blocks
{ $Tproc(P_i,k1)$ - $Tproc(P_i,pcmin)$ }; pcmin is min [pc, pc']
This expression models the worst case situation for evaluating $Texec(P_i)$.

The expression $Tproc(P_i,p)$, for any operation $P_i$, is given by the following:

$$\sum_{i=1}^{i=p} [\ Tin\text{-}R1_i + Tin\text{-}R2_i + Tcomp_i + Tout_i\ ]$$

The expression $Tprod(P_i,p)$, for any operation $P_i$ at level 1 is given by

$$\sum_{i=1}^{i=p'} [\ Tin\text{-}R1_i + Tin\text{-}R2_i + Tcomp_i + Tout_i\ ]$$

where p' is the number of input blocks consumed to produce p blocks of output. For subsequent levels,

$Tprod(P_i,p)$ = max [ $Tprod(P_{i-1},p')$, $Tprod(P_{i-1}',p')$ ]

We study the performance of this algorithm with and without pipelining, with respect to three parameters. The first parameter is the block size, B, which we vary from 20 to 100. The value of Tbr will also vary correspondingly. The second parameter is the join selectivity, j, of the critical path operations and the operations that produce answers. The join selectivity is not an absolute value but is defined as a ratio; thus we vary j in proportion to the sizes of the input relations for each operation. The third parameter is the number of tuples, N, of the initial database relation, noted A-ff. We vary this parameter from 200000 to 800000.

Figure 5.2 shows the response time and the execution time for both the pipelined and distributed cases as a function of the depth i of the resolvents, Ti-bb. In this plot, the value of N is 200000, B is 20 and j is $5*10^{-6}$. This value of j (=1/N) ensures that the size of the output produced by the critical operations is roughly equal at different levels along the pipeline. The figure shows that for small i, with pipelining, the response time is much less than the execution time. As i increases these two curves tend to move closer. The reason is that for small i there are less operations (and delays) along the critical path and thus, the response time is small. As i increases, there are more operations (and delays) along the critical path which tend to increase the delay in producing the first block of output for Ti-bb. The figure also indicates that for small i, the execution time for the distributed and pipelined approaches are very close but these curves tend to diverge with increasing i. This is because for small i, there are fewer operations in the pipeline and the advantage of pipelining on the execution time is limited. As i increases, the number of operations in the pipeline increase and the performance improvement increases correspondingly.

In Figure 5.3, we show the effect of the block size, B, (or operand granularity) on the response time. We plot the ratio of the response time in the distributed case to the response time with pipelining, as a function of the block size. We examine three resolvents, T4-bb, T6-bb and T8-bb. This ratio, which is proportional to the performance improvement due to pipelining, is largest for B=20, and gradually decreases with increasing block size. This is true for all resolvents. The reason is that the response time is closely related to the block size. The number of tuples consumed to produce a small block of output is less, and this reduces the response time. As the block size increases, more input tuples have to be consumed to produce the first block and the response time increases.

Figure 5.4 shows the effect of block size on the execution time. For resolvents T24-bb and T32-bb, we plot the execution time of the distributed case and the pipelined case, as a function of B. The execution time, which is the time to complete processing all blocks, is not as sensitive to pipelining as the response time. However, with increasing values for B, the execution time reduces slightly. This is because the delays along the critical path seem to be larger for smaller block size, i.e., with smaller block sizes, the input rate controls (and delays) the pipeline rate along the critical path for a longer time. This delay has a corresponding effect on the execution time and the execution time is slightly less with larger block size. If we assumed a penalty for transmitting smaller blocks, then the execution time for the distributed case would also reduce slightly, with larger block size.

Figure 5.5 shows the effect of the join selectivity j, on the response time, for various resolvents. The ratio of the response time in the distributed case to the response time with pipelining is plotted as a function of j. This ratio is seen to increase with increasing join selectivity for all resolvents. The reason is that with increasing values of j, less input blocks have to be consumed to produce the first block of output. As a result, the response time for the pipelined case is smaller.

Finally, in Figure 5.6, we show the effect of N, the number of tuples of the database relation, A-ff, on the execution time for the distributed and pipelined cases. Note that the value of join selectivity is varied correspondingly; this ensures that the size of the answers produced for each resolvent is proportional to the size of the input relations and the analysis is unbiased by arbitrary sizes of the output relations. The execution time for two resolvents T24-bb and T32-bb are shown. Each of these plots is linear in N, as is expected since the execution time must be proportional to the size of input relations being processed. However, the execution time for the distributed case increases more rapidly with increasing N as compared to the pipelined case, resulting in enhanced performance due to pipelining. The reason is that after the initial delays in setting up the pipeline, the longer the piepeline operates under steady state, the greater the benefit of pipelining. With increasing N, the pipeline operates longer under steady state, hence the improved performance.

## 6. Conclusions and Future Research

To summarize, we presented a strategy for the concurrent evaluation of the resolvents generated by a recursive query using query processing and optimization techniques. Analytical formulae were derived for the response times and execution times of the concurrent queries and the performance gain due to pipelining was examined.

To summarize the results of our analysis, the pipelined approach with both vertical and horizontal concurrency always performed better than the distributed approach, which uses only horizontal concurrency; both approaches used intermediate result sharing. The effects of pipelining on the response time is much more pronounced as compared to the execution time; this is as expected since pipelining inherently produces data at an earlier instant. The effects of the block size, B, and the join selectivity, j, on the response time are similarly explained. The advantages of pipelining are more marked with longer sequences of operations in the pipeline, e.g., with increasing depth i, of the resolvents, Ti-bb. The advantages of pipelining are also greater, the longer the pipeline operates under the steady state (after the initial delays), e.g., with larger database relations.

The example of transitive closure is a direct recursion. In [NAQ84] an indirect recursion is described for S, as follows:

S(x1,z1) :- M(x1,y1), T(y1,z1)
T(y1,z1) :- S(y1,w1), P(w1,z1)
T(y1,z1) :- F(y1,z1)

where M, F and P are database relations. A query of the form S(c,?), or S-bf, would generate the set of resolvents, Si-bf, of Figure 6.1 and, by applying the method described in Section 4, an evaluation strategy such as shown in Figure 6.2 can be obtained.

In general, any recursive intension can be reduced to the form:

S(.,.,..) :- M(.,.,..), S(.,.,..), P(.,.,..)

where M and P are relational algebra expressions. We are currently developing an algorithm, based on the method described in section 4, to evaluate any general recursive intension S. Mutually recursive clauses may require further research.

## 7. REFERENCES


AST76 Astrahan, M.M., et. al., "System R: A Relational Approach to Database Management," *ACM TODS*, 1,2, 1976.

BIC81 Bic, L. et.al., "An Architecture for a Relational Dataflow Machine," *Proc. SIGMOD Symp. on Small Systems*, 1981.

BOR80 Boral, H., and DeWitt, D.J., "Design Considerations for Dataflow Database Machines," *ACM/SIGMOD*, 1980.

BOR84 Boral, H. et.al., "A Methodology for Database System Performance Evaluation," *Proc. ACM/SIGMOD*, 1984.

BRO84 Brodie, M.L. and Jarke, M., "On Integrating Logic Programming and Databases," *Proc. 1st Workshop on Expert Database Systems (FIWEDS)*, 1984.

CHA81 Chang, C., "On the Evaluation of Queries Containing Derived Relations in a Relational Databases," in *Advances in Data Base Theory*, Vol. 1, 1981.

CHO85 Chou, H. et.al., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *VLDB*, 1985.

DEW84 DeWitt, D.J., et. al., "Implementation Techniques for Main Memory Database Systems," *Proc. ACM/SIGMOD*, 1984.

FIN82 Finkelstein, S., "Common Expression Analysis in Database Applications," *ACM/SIGMOD*, 1982.

GAL83 Gallaire, H., "Logic and Databases," *IJCAI Panel*, 1983.

HAN86 Han, J. and Lu, H., "Some Performance Results of Recursive Query Processing in Relational Database Systems," *Proc. Conf. on Data Engg.*, 1984.

HON84 Hong, Y.C., "A Pipelined Architecture for Relational Database Operations," *Proc. Conf. on Data Engg.*, 1984.

JAR84 Jarke, M., et. al., "An Optimizing Prolog Front-end to a Relational Query System," *Proc. ACM/SIGMOD*, 1984.

KEL82 Kellogg, C., "Knowledge Management: A Practical Amalgam of Knowledge and Data Base Technology," *Proc. AAAI*, 1982.

KER84 Kerschberg, L. and Shepherd, A., "PRISM: A Knowledge Based System for Semantic Integrity ...," *Proc. ACM/SIGMOD*, 1984.

KIM85 Kim, W., Gajski, D. and Kuck, D., "A Parallel Pipelined Relational Query Processor," *ACM TODS, 9,2*, 1984.

MIK85 Mikkilinenni, K., "Distributed Query Processing Techniques based on Pipelining and Data Sharing ...," Ph.D. thesis, University of Florida, 1985.

MIN81 Minker, J. and Nicolas, J.M., "On Recursive Axioms in Relational databases," Univ. of Maryland Tech. Rep, 1981.

NAQ84 Naqvi, S.A. and Henschen, L.J., "On Compiling Queries in Recursive First Order Databases," *JACM, 31,1*, 1984.

RAS85 Raschid, L. and Su, S.Y.W., "Capturing Semantic Knowledge in an Integrated Knowledge Base," Univ. of Florida, Tech. Rep., 1985.

REI78a Reiter, R., "Deductive Question Answering on Relational Data Bases," in *Logic and Databases, 1978.*

REI78b Reiter, R., "On Closed World Databases," in *Logic and Databases*, Plenum Press, 1978.

ROB65 Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle," *JACM, 27,2*, 1965.

ROT80 Rothnie, J.B., et. al., "Introduction to a System for Distributed Databases (SDD-1)," *ACM TODS, 5,1*, 1980.

SIC76 Sickel, S., "A Search Technique for Interconnection Graphs," *IEEE Trans. on Computers, 1976.*

STO76 Stonebraker, M., et. al., "The Design and Implementation of INGRES," *ACM TODS 1,3*, 1976.

SU85 Su, S.Y.W. and Raschid, L., "Incorporating Knowledge Rules in a Semantic ...," *IEEE Conf. on AI Appl.*, 1985.

SU86a Su, S.Y.W. and Mikkilinenni, K., "A Distributed Query Processing Strategy ...," *Proc. Conf. on Data Engg.*, 1986.

ULL85 Ullman, Jeffrey D., "Implementation of Logical Query Languages for Databases," *Proc. ACM/SIGMOD*, 1985.

WIE84 Wiederhold, G. and Missikoff, M., "Towards a Unified Approach to Expert and Database Systems," *FIWEDS*, 1984.

WON76 Wong, E., et. al., "Decomposition: A Strategy for Query Processing," *ACM TODS, 1,3*, 1976.

## ACKNOWLEDGEMENT

This research is partially supported by a grant from the National Bureau of Standards, #60NANB4D0017.

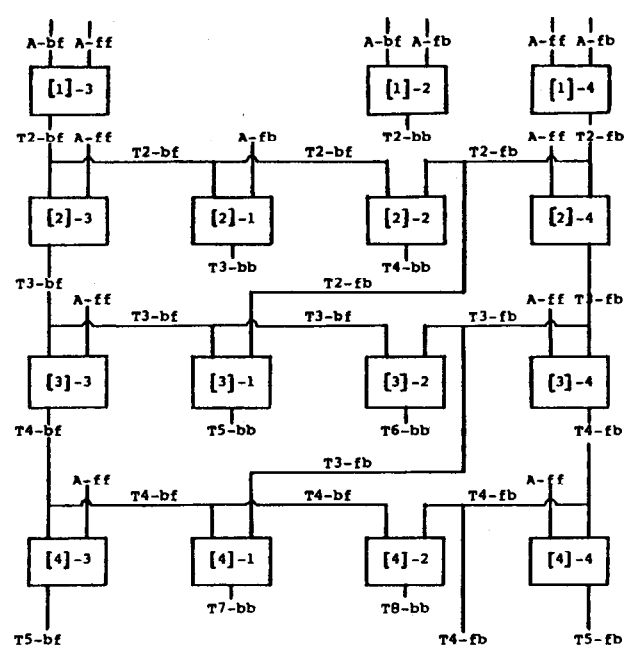


Figure 4.3 Architecture for Evaluating Transitive Closure: Parallel Operations, Interconnections and Data Flow

| | |
|---|---|
| T1-bb | A(a,c) |
| T2-bb | A(a,y1), A(y1,c) |
| T3-bb | A(a,y2), A(y2,y1), A(y1,c) |
| T4-bb | A(a,y3), A(y3,y2), A(y2,y1), A(y1,c) |
| T5-bb | A(a,y4), A(y4,y3), A(y3,y2), A(y2,y1), A(y1,c) |
| T6-bb | A(a,y5), A(y5,y4), A(y4,y3), A(y3,y2), A(y2,y1), A(y1,c) |
| : | : |

Figure 4.1 Expressions Corresponding to Resolvents Ti-bb

| | |
|---|---|
| | [1]-1 |
| T1-bb | A-bb |
| | [1]-2 |
| T2-bb | (A-bf JN A-fb) |
| | [1]-3 [2]-1 |
| T3-bb | ( (A-bf JN A-ff) JN A-fb) |
| | common [2]-2 [1]-4 |
| T4-bb | ( (A-bf JN A-ff) JN (A-ff JN A-fb) ) |
| | common [2]-3 [3]-1 common |
| T5-bb | ( ( (A-bf JN A-ff) JN A-ff) ) JN (A-ff JN A-fb) ) |
| | common common [3]-2 [2]-4 common |
| T6-bb | ( ( (A-bf JN A-ff) JN A-ff) ) JN (A-ff JN (A-ff JN A-fb) ) ) |
| : | : |
| : | : |

Note: "common" identifies a sub-expression that has been evaluated

Figure 4.2 Identifying Parallel Primitive Operations and Common Sub-expressions in Evaluating Resolvents Ti-bb

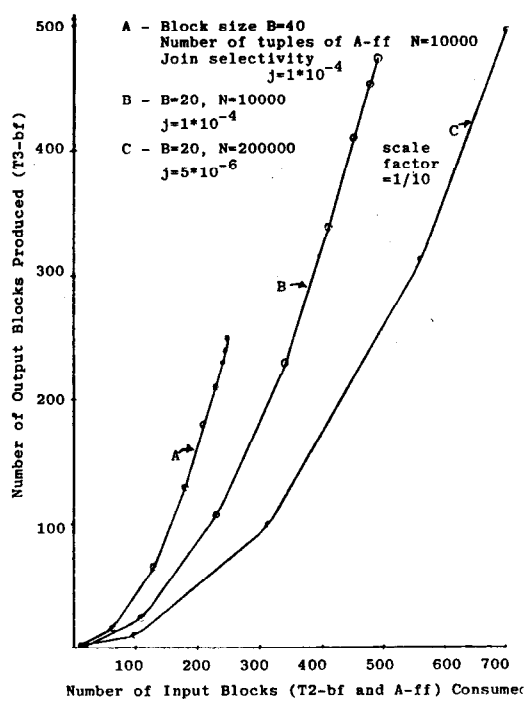


Figure 5.1 Rate of Output Pipeline of Operation [2] -3

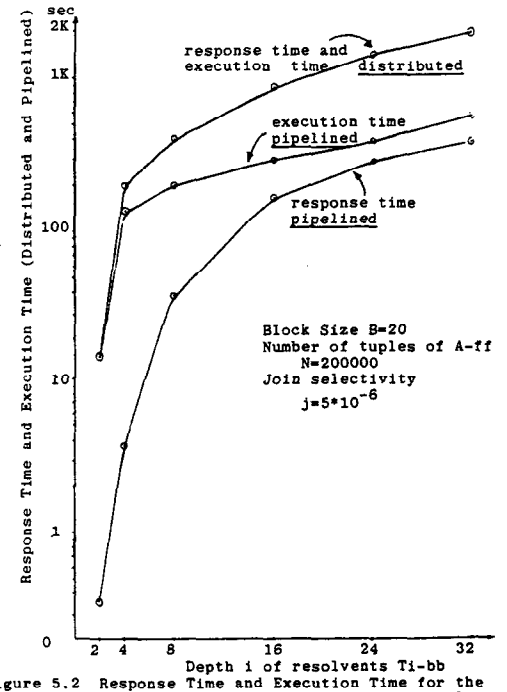


Figure 5.2 Response Time and Execution Time for the Distributed and Pipelined Cases vs. Depth of Resolvents

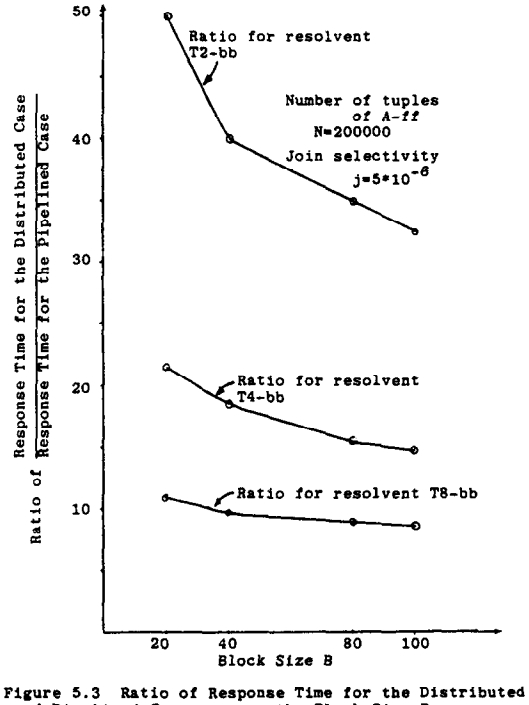


Figure 5.3 Ratio of Response Time for the Distributed and Pipelined Cases versus the Block Size B

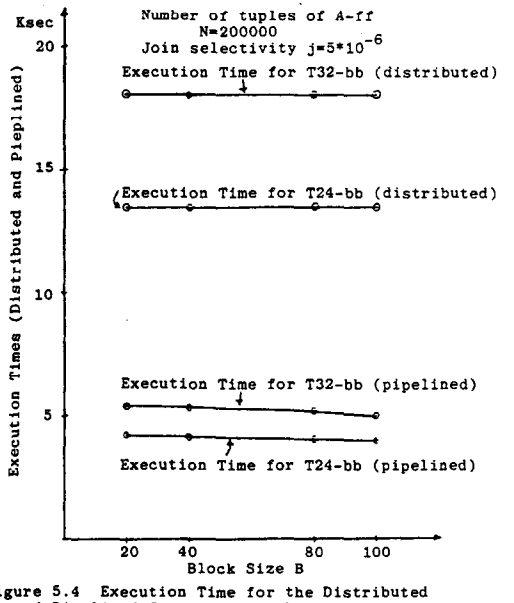


Figure 5.4 Execution Time for the Distributed and Pipelined Cases versus the Block Size B

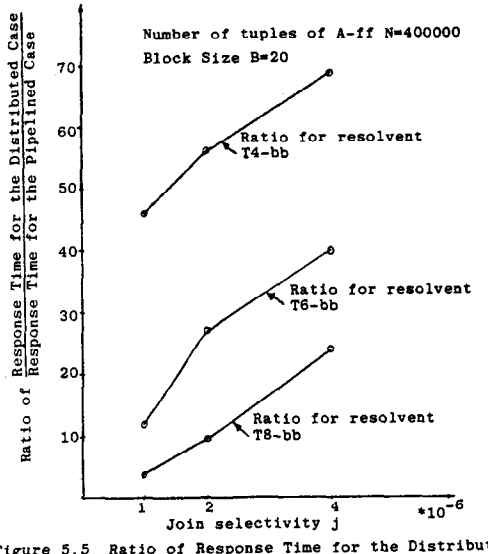


Figure 5.5 Ratio of Response Time for the Distributed and Pipelined Cases versus Join Selectivity j

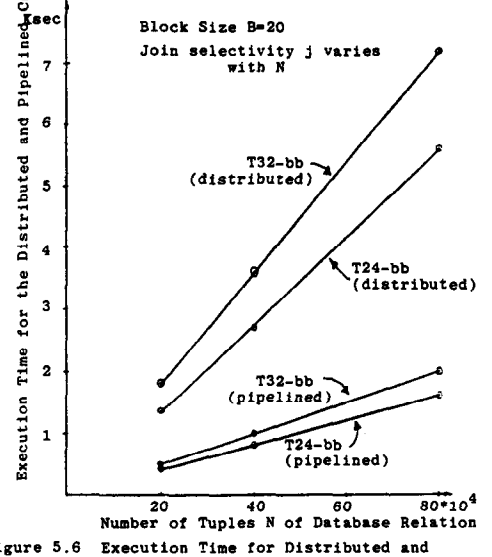


Figure 5.6 Execution Time for Distributed and Pipelined Cases vs. Number of Tuples of Database Relation A-ff

S1-bf M(c,y1), F(y1,?)

S2-bf M(c,y3), M(y3,y2), F(y2,y1), P(y1,?)

S3-bf M(c,y5), M(y5,y4), M(y4,y3), F(y3,y2), P(y2,y1), P(y1,?)

: :

Figure 6.1 Expressions Corresponding to Resolvents Si-bf

S1-bf [1]-1
(M-bf JN F-ff)

S2-bf [1]-3 [2]-1 [1]-2
( (M-bf JN M-ff) JN (F-ff JN P-ff) )

S3-bf common [2]-3 [3]-1 [2]-2 [1]-4
( ( (M-bf JN M-ff) JN M-ff) JN (F-ff JN (P-ff JN P-ff) ) )

S4-bf common common [3]-3 [4]-1 [3]-2 [2]-4 common
((((M-bf JN M-ff)JN M-ff) JN M-ff) JN (F-ff JN (P-ff JN(P-ff JN P-ff))))

: :

Figure 6.2 Identifying Parallel Primitive Operations and Common Sub-expressions in Evaluating Resolvents Si-bf