A Message Passing Framework for Logical Query Evaluation

Allen Van Gelder *
Stanford University

Abstract

We formulate the problem of query evaluation over a relational database (EDB) supplemented by function-free Horn clause rules (IDB) as a system of cooperating processes communicating by message passing Shared memory is not required, making this approach suitable for distributed systems This modularization offers flexibility of implementation and opportunities to use existing operating system features to enhance performance, as well as providing a step toward practical parallel implementation The technique is based on top-down construction of a rule/goal graph followed by a mixed top-down and bottom-up evaluation strategy employing "sideways information passing" to restrict the computation to relevant, or at least potentially relevant, portions of intermediate relations Termination is guaranteed, but the termination conditions are global in general, and their distributed asynchronous detection requires care We describe an efficient protocol to accomplish this We describe a greedy information passing strategy and introduce the monotone flow property for rules We discuss the relation of these concepts to acyclic database schemas and qual trees

*Supported by NSF grant IST-84-12791 and a grant of IBM Corp

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery To copy otherwise, or to republish, requires a fee and/or specific permission

© 1986 ACM 0-89791-191-1/86/0500/0155 $00 75

1 Problem Statement

We consider a first-order logical system without function symbols consisting of

- An *extensional database (EDB)* consisting of ground atomic formulas, also called *facts*, the EDB may be viewed as a conventional relational database
- A *permanent intentional database (PIDB)* consisting of a set of Horn clauses, also called *rules*, which contains no positive occurrence of a predicate that appears in the EDB, and contains no occurrence of the distinguished predicate, **goal** The positive literal of a clause is called the *head* of the rule and the negative literals are called its *subgoals*
- A *query* consisting of a set of Horn clauses (frequently just one) whose positive literal is a distinguished predicate, **goal**, which does not appear negatively anywhere
- The union of the PIDB and the query rules is called the *intentional database (IDB)*

We wish to compute the **goal** portion of the minimum Herbrand model of this system, i e , we wish to find all tuples t such that **goal**(t) is logically implied by the union of the IDB and the EDB Hereafter, we abbreviate "Herbrand model" to "model" We interpret this relation as the answer to the query, and call the computation *evaluating* the query

The efficiency issue can be summarized as restricting the computation as far as possible to the portion of the minimum model that contributes to the derivation of **goal** tuples We can view a bottom up computation as an operator on the combination of the IDB and EDB facts that takes as input all facts derived in n or less steps and produces all facts derived in $n+1$ steps In the nonrecursive case, this can be reduced to an operator on the EDB alone whose output is **goal** facts [Rei78] This operator

takes the form of a relational calculus or relational algebra expression, and restriction to relevant tuples is accomplished (at least in part) by known optimization techniques [Ull82] Such a reduction is not possible in the recursive case The need to compute even a specified fact $p(a, b)$ can give rise to subgoals $p(a, Y)$ or even $p(X, Y)$ where the needed values of X and Y are not known a *priori* Therefore, restricting the portion of the minimum model that is to be computed, if it is done at all, must be done *during* the computation

1.1 Previous Work

The problem was formulated, and the case of nonrecursive rules was considered, by Reiter [Rei78], who showed that this case can be handled by standard relational algebra methods [Ull82] In the recursive case, the solution amounts to a least fixed point computation, as was observed in [VEK76,AU79], although no actual algorithms were presented there

It is known that the recursive problem can be solved by brute force, essentially by enumerating all possible ground instances of the IDB with all possible combinations of constants that appear in the system (primarily in the EDB) substituted for the variables, and "reasoning forward" until the minimum model is derived The running time is $O(n^{k+O(k)})$ if there are n constants in the system and at most k variables in any rule Vardi has shown [Var82] that for a given EDB, there is a worst-case exponential lower bound on query evaluation time as a function of the length of the IDB, provided that an arbitrary number of variables per rule is permitted So in a sense the brute force bound is tight for the most general case However, in practice, rules do not have a huge number of variables, and exhibit other regularities of behavior, so it is useful to look for special cases that can be solved more efficiently

Henschen and Naqvi [HN84] have reported a method that tries to deal efficiently with a class of these problems, the primary limitation being *linear recursion*, which means that the head of any rule is recursively related to at most one subgoal in the same rule

McKay and Shapiro [MS81] have reported a more general solution in which the entire minimum model is not computed, but intermediate relations that are needed tend to be entirely computed, even if only a small part is actually useful for answering the query

Walker has described a system called Syllog [Wal81] The method is described informally, so it is unclear how general it is

Ullman has proposed a formalism using "rule/goal graphs" and "capture rules" for analyzing a wider class of query evaluation problems including rules with function symbols [Ull84] The focus of that work is on choosing among general computation methods, a capture rule says "if the problem (or subproblem) has such-and-such properties, then such-and-such a method is applicable "

Porter has developed an approach based on Earley parsing [Por85] Recent proposals for treating this problem can also be found in [Loz85,Vie85]

1.2 Summary of Results

We formulate the problem as a distributed computation in a network of processes communicating by messages The network structure depends only on the IDB Each process computes an intermediate relation, more or less by standard relational algebra methods During an initialization phase the specifications for the relation are received in messages from neighboring processes, and the process generates messages requesting the relations it needs as input During the computation phase additional specifications (in the form of partial bindings) and result tuples are passed back and forth by messages Finally, termination messages signal that the requested relations have been completed No shared memory is required, however, shared memory for intermediate relations can be used to reduce the total space requirement, at a sacrifice in simplicity Thus this formulation is amenable to parallel computation

We describe a procedure that "builds" the network as a rule/goal graph that essentially reflects the structure of the IDB, and show that the procedure terminates for any finite IDB

An important feature of the rule/goal graph is that predicate arguments are divided into 4 classes, "c," "d," "e," and "f" Class "c" are constants known at graph-construction time Class "f" are free variables, the job is to find bindings for them Class "e" are free variables whose values are not used, only the *existence* of a value is material Class "d" is the important class for efficiency, especially in recursively defined relations, an argument in this class is bound during the computation to a *set* of needed values Thus a class "d" argument functions as a semi-join operand (Perhaps think of "d" as standing for "dynamically bound ") It serves to restrict the computed part of the intermediate relation to values that are (at least potentially) useful for deriving goal tuples

We present a basic set of messages that drive the computation, and are sufficient to carry out the computation of the query answer The basic set can

be extended in order to pass optimization information, offering the possibility of taking advantage of statistics on the EDB and using various heuristics

Recursive rules produce cycles of messages In particular, this method handles *nonlinear recursion*, in which a goal depends recursively on two or more of its subgoals in the same rule Nonlinear recursion frequently arises in divide-and-conquer algorithms, and occurs in certain polynomially complete problems [Var82] Deletion of duplicates in cycles ensures that nodes become idle when the computation is complete An additional message protocol allows asynchronous distributed detection of the fact that all nodes in a cycle are idle simultaneously

We introduce the *monotone flow property* for rules, and argue that for such rules there is an efficient way to compute intermediate relations in the absence of other information Our notion of efficiency involves avoiding the generation of tuples that do not contribute to the query answer Rules that do not satisfy the monotone flow property have an inherently cyclic structure that can produce intermediate results that are much larger than the final results, even when the subgoals' relations are pairwise consistent, i e , pairwise, they have no "dangling tuples " Experience indicates that the majority of rules that occur in practice have the monotone flow property, so developing a methodology to take advantage of it is justified

These results have several possible applications They enhance the power of logic programming interpreters by freeing the user from specifying the order in which rules and subgoals within rules are processed, this is analogous to the role of the relational database model in freeing users from specifying navigational information In addition, the method is certain to terminate, avoiding the well-known "left recursion" problems of strictly top-down methods Finally, the rule/goal graph with message passing between nodes breaks the problem up into several modules with well defined interfaces This decomposition offers opportunities to use existing operating system features, such as scheduling, message queueing, and multi-tasking It also provides a natural approach to parallel implementation of logical query evaluation

2 Rule/Goal Graph Construction

2.1 Basic Rule/Goal Graphs

In order to build the basic rule/goal graph corresponding to the IDB, we begin by constructing a tree of goals reduced to subgoals through rules, much in the manner of Prolog and other top-down systems, starting with a top-level *goal node* for goal Depth first search is used, and terms "parent," "child," "ancestor," and "descendant" will refer to the depth first spanning tree induced We consider edges in this tree to be oriented from child to parent, the direction in which "answers" flow Whenever an EDB subgoal (i e , a subgoal whose predicate symbol is in the EDB) is created, it remains as a leaf, it is not processed against the actual EDB relation during graph construction Furthermore, whenever an IDB subgoal is a variant of one of its ancestors, instead of expanding it further, we just create a *cyclic edge*[1] from the appropriate ancestor to the variant subgoal When these exceptions do not apply, we expand a subgoal by creating a *rule node* for every rule whose head unifies with the subgoal and connect an arc from the subgoal node to each such rule node The rule node contains a copy of the rule that began with all new variables, then had the most general unifier (mgu) applied Thus the head in the rule node is exactly the same as the subgoal of its parent Now we create new *goal nodes* for all subgoals of the new rule node and put edges from the rule node to its subgoals

Strong components in the rule/goal graph play an important role in the computation Recall that a strong component is a maximal set of nodes such that each has a path to the other The *reduced graph* is obtained by collapsing each strong component to a single node, and is acyclic We also use the following terminology

Definition 2.1: Let $r \rightarrow s$ be an arc in the rule/goal graph As usual, we call r a *predecessor* of s, and call s a *successor* of r Furthermore, if r and s are in different strong components, we say r is a *feeder* of s, and s is a *customer* of r □

2.2 Information Passing

Definition 2.2: An *information passing rule/goal graph* is a variation of the basic rule/goal graph, but is a more elaborate form, in which we propagate argument classes "c," "d," "e," and "f" from the goal (i e , parent) of the rule node to its subgoal nodes Before creating a cyclic edge to a goal node from an ancestor, we now require not only that the ancestor is a variant, but that the arguments match on their classes as well □

The class letters are mnemonics for "constant," "dynamic," "existential," and "free," respectively

[1] In the terminology of depth first search, this is a *back edge* This term is clear in a context where tree edges are directed from parent to child, however, in our context, it would be confusing to use this terminology

Classes of arguments containing a variable that appears in the goal are simply passed through to the corresponding subgoal

The subgoal arguments whose variables do not appear in the goal are classified as either "d" or "f" according to an *information passing strategy*, assuming the variable appears in more than one subgoal Any such variable appearing in multiple subgoals may be classified as "d" in all but one of those subgoals The idea is that the subgoal(s) that retain the "f" designation will be evaluated first and will furnish a set of valid values for that argument (hence variable) to the rule node, and the rule node will pass them to subgoals that have "d" designations The term "sideways information passing" was coined in [Ull84] to describe this technique This is similar to, but more general than, the technique used in Prolog and other top-down systems to bind subgoal arguments Essentially, Prolog solves the subgoals in order, left to right Here the system decides in which order to solve them, and can even choose to solve some "simultaneously"

Definition 2.3: More formally, an *information passing strategy* is an acyclic directed graph on the subgoals of a rule The arc $r \rightarrow s$ is present whenever an "f" argument of r furnishes bindings for a "d" argument of s □

Definition 2.4: A *greedy* information passing strategy is one in which the set of "d" arguments in the subgoals is maximally pushed forward, in other words, no subgoal relation is requested with some argument free if we could wait for tuples from another subgoal that has more bound arguments, and provide a set of bindings for that argument □

Greedy information passing is based on the heuristic that maximizing bound arguments is more important than minimizing unbound arguments for the purpose of making intermediate relations small

If a variable appears in one subgoal and nowhere else in the rule, it is labeled "e" It could be treated as "f" and produce correct results, but the "e" designation indicates that its value will not be transmitted, possibly permitting greater efficiency For example, goal $p(X^f, Y^e)$ can be satisifed by producing one tuple for each unique X even though there may be many Y values that go with a given X

Example 2.1: Consider an example program $\mathbf{P}_1$ containing two EDB relations, r and q, an IDB predicate p, and query $p(a, Z)$, where a is a constnat entered by the user The IDB consists of the following Prolog-style rules, where the goal p is on the left and the subgoals are on the right (Read "←" as "if")

$$\begin{array}{ll} \mathbf{P}_1 & \mathrm{goal}(Z) \leftarrow p(a, Z) \\ & p(X, Y) \leftarrow p(X, U),\ q(U, V),\ p(V, Y) \\ & p(X, Y) \leftarrow r(X, Y) \end{array}$$

The greedy information passing strategy for the recursive rule is

$$p(X, U) \rightarrow q(U, V) \rightarrow p(V, Y)$$

since only X is bound initially The information passing rule/goal graph corresponding to this strategy is shown in Fig 1 (We omit the two top levels of the graph, involving **goal**, as they are trivial in this case) Requests for information flow against the direction of the edges, "answers" flow in the edge direction Cyclic edges are shown by dashed lines The superscripts on variables denote the binding status of the arguments in which they appear Observe that the goal node $p(a^c, Z^f)$ cannot supply tuples to nodes with different binding patterns, necessitating a separate goal node for $p(V^d, Z^f)$, however, a change in variable name does not prevent a goal node from supplying tuples to another node with the same binding pattern Thus $p(V^d, Z^f)$ supplies tuples to $p(V^d, Y^f)$ and $p(W^d, Z^f)$ in response to requests from those nodes, each request for tuples from $p(V^d, Z^f)$ provides a *set* of bindings for the first argument

The evaluation of the recursive rule for $p(V^d, Z^f)$ in this graph proceeds as follows The leftmost subgoal inherits a "d" binding from the head, i e , a set of values for the first argument The first step is to determine what p tuples match those values Then the the second arguments of these p tuples are used as a set of bindings for the first argument of q, the second subgoal The set of second arguments for q becomes the set of first arguments for the third subgoal, the rightmost p Finally, the set of values for the second argument of the right p are associated with the original bindings, producing answers However, since p is recursive, all steps are interleaved □

Theorem 2.1: The information passing rule/goal graph construction procedure terminates for any finite set of function-free IDB rules, and the size of the graph is independent of the sizes of the EDB relations

Proof There are only a finite number of predicate symbols, argument positions, and permutations of bindings, and hence only a finite number of goal node labels such that no two are variants This provides a bound on the length of a simple path, which together with finite degree of nodes, guarantees finiteness of the entire graph There is a technicality that repeated

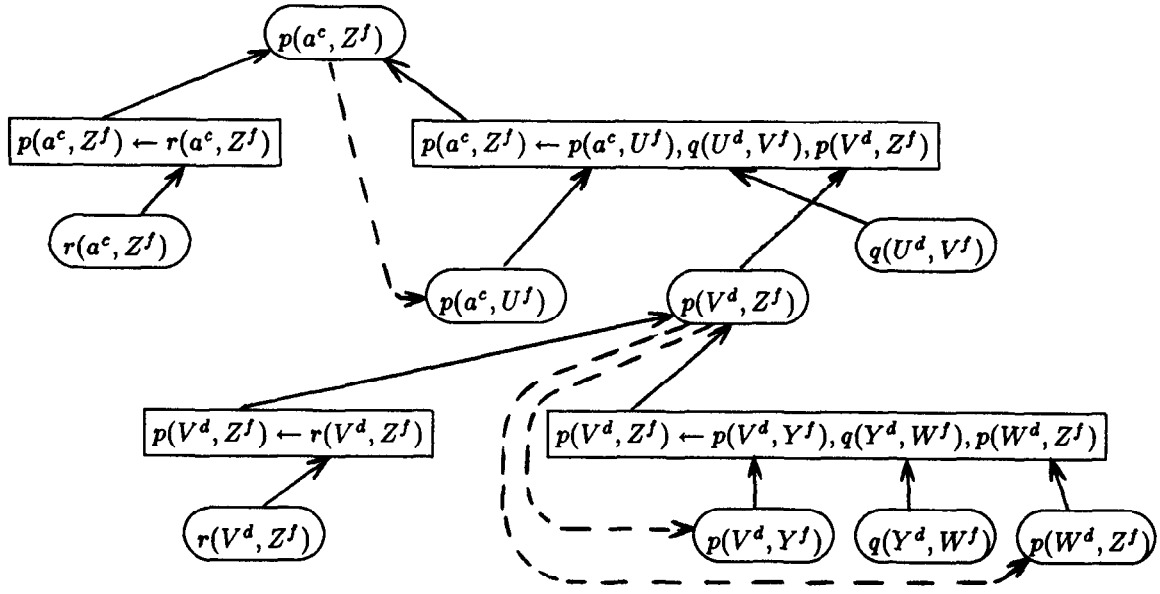


Figure 1 Greedy information passing rule/goal graph for $\mathbf{P}_1$

variables can occur in various patterns in different nodes, such as $p(X, X, Z)$ and $p(U, V, V)$, preventing them from being variants, but again, there are only a finite number of such patterns ∎

When the graph is finished, we interpret each node as a processor that performs a relational computation Predicate nodes with rule-children compute the *union* of the relations computed by their children, rule nodes combine their subgoal relations using *join, select, and project* The predicate nodes that are connected to an ancestor predicate node by a cyclic edge perform a selection on the relation computed by the ancestor

Observe that several nodes in the graph may have identical predicates and binding patterns For single processor computation it is probably desirable to coalesce such nodes (thereby introducing *cross* and *forward edges*) However, for distributed or parallel computation, combining nodes may well be counter-productive, so in this paper we shall assume that it is not done

3 Message Controlled Computation

We now briefly outline how messages are used to accomplish a distributed query evaluation, once the appropriate graph is "built "

3.1 Basic Messages

The *relation request* message triggers the beginning of computation and identifies the classes of the arguments It originates at the top level goal node and is passed through the graph, *against* the orientation of the arcs Note that goal nodes with cyclic arcs out of them will receive at least two requests

When a relation request includes some class "d" arguments, there follow a series of *tuple request* messages as the computation proceeds Each tuple request message specifies one binding for all of the "d" arguments [2] Thus the complete specification of an intermediate relation consists of the relation request and the set of associated tuple requests

Whenever a tuple is derived it is sent to the parent via a *tuple* message In the case of a goal node with cyclic edges out of it, the *tuple* message is also sent to the other successor nodes, which are descendants Processes do not block, waiting for complete answers, before issuing new requests In fact, in recursive rule evaluation, it would be impossible to do so

[2] A further enhancement would be to "package" a set of related tuple requests, in case the node servicing the request can gain some efficiency of volume E g , if an EDB relation $r(X,Y)$ has no index on its second argument, then tuple requests $r(X,a)$, $r(X,b)$, , presented separately require the whole r relation to be scanned for each one If packaged, the retrieval can be done in one scan of r Considering that an index can be built with one scan of r, and in the interest of simplicity, we omit this feature

Therefore, answer tuples corresponding to different tuple requests may be interleaved

There are various trade-offs between the amount of space used to store temporary relations, the amount of communication between processes, and the amount of redundant computation in the form of joins and database retrievals We shall confine ourselves to a method that assumes space is plentiful, but communication is expensive, because that seems to be the simplest to describe and verify Variations can be defined and justified in relation to this basic method Since tuples and tuple requests come trickling in throughout the computation, it is appropriate for rule nodes to store their subgoals' temporary relations, assuming no shared memory When a tuple arrives, provided it does not duplicate one already received, it is matched against the (partial) temporary relations of other subgoals to form new tuples via joins Detection of duplicates is necessary to allow loops to terminate In addition, goal nodes store their temporary relations, and only forward answer tuples that are genuinely new (Trivial goal nodes, with only one in-edge and one out-edge are exempt) A goal node with multiple out-edges needs to furnish answers in separate streams to each successor node, different successors normally will be out of synchronization, and normally will have requested different subsets of the total temporary relation at the goal node

3.2 Distributed Termination of Cycles

When a feeder node determines that it can produce no more tuples for a particular tuple request (or relation request), it sends an *end* message to notify its customer (parent) of that fact A prerequisite is reception of *end* messages from all its own feeder subgoals The tricky part is determining asynchronously that a recursively defined relation is finished An additional protocol is used to determine that all queues in the strong component are empty The problem is that one (or a few) answer tuples may be trickling through the nodes of the strong component, yet each node happens to be caught up on its work at the time the message arrives asking whether it is done The solution is to designate the unique feeder node of each strong component as the "BFST leader," and define a breadth first spanning tree (BFST) for that strong component [3] The protocol is summarized in pseudo-code in Fig 2 and described below

The boolean function *empty-queues*() is intended to return **true** when the node has received *end* messages from all its feeders, and is itself idle When *empty-queues*() is true for the BFST leader, it originates an *end request* message This message is propagated through the BFST to all nodes of the strong component, against the direction of the edges (i e , in the normal direction for request messages)

Each node, upon receiving an *end request* message, "remembers" its arrival and sends an *end request* message to all its "child" nodes in the spanning tree The leaves of the spanning tree automatically answer the first *end request* with an *end negative* message, which is duly passed up the BFST to the leader Internal nodes of the BFST pass an answer up only after receiving an answer from all children Upon receiving an *end negative* answer, and assuming *empty-queues*() is still true, the BFST leader starts another *end request* message down the BFST, and repeats this after each *end negative* answer If a node has been idle for the entire period between two *end requests*, and has received an *end confirmed* message from all its children in the BFST, it answers the latest *end request* with an *end confirmed* message Otherwise, after receiving answers from all children, it continues to answer *end negative* In order to "remember" how long it has been idle, the node increments the variable *idleness* if it is idle upon receiving an *end request* (based on *empty-queues*() being true), it resets *idleness* to zero whenever it receives work

If the BFST leader receives *end confirmed* from all its children and has itself been idle since its last *end request*, then it concludes the protocol by sending an *end* message to its customer (parent) node [4]

Theorem 3.1: In the protocol described above and shown in Fig 2, the BFST leader issues an *end* message if and only if all nodes in the strong component are idle and *end* messages have been received from all feeders of the strong component

Proof If a node responds *end confirmed*, then its *idleness* is at least 2, meaning it has been idle for the period between the two most recent *end request* messages Therefore, all nodes in the strong component are idle at the time the leader originates the last *end request* message, i e , the one to which all nodes respond *end confirmed* Furthermore, no node responds *end confirmed* unless it has received *end* messages from all its feeders ∎

[3] The absence of *cross* and *forward* edges guarantees a unique leader, and also ensures that the BFST coincides with the depth first spanning tree

[4] If nodes with identical predicates and binding patterns were coalesced, then the leader must propagate the *end* message around the strong component, as other nodes may have customers

```
boolean: empty-queues();
integer: idleness := 0;
...
procedure send-answer-tuple
    ...
    if BFST-leader and empty-queues() then
        idleness := 1;
        create-end-request;
        process-end-request;
    end if;
end proc;

procedure process-end-request
    if empty-queues() then
        idleness := idleness + 1
    else
        idleness := 0
    end if;
    waiting-for := num-BFST-children;
    if waiting-for > 0 then
        for child in BFST-children do
            send-end-request(child);
            idleness := empty-queues();
        end for
    else
        if idleness > 1 then
            send-end-confirmed(BFST-parent)
        else
            send-end-negative(BFST-parent)
        end if
    end if;
end proc;
```

```
procedure process-end-negative
    waiting-for := waiting-for −1;
    if waiting-for = 0 then
        if BFST-leader then
            if empty-queues() then
                idleness := 1;
                create-end-request;
                process-end-request;
            end if
        else
            send-end-negative(BFST-parent)
        end if
    end if;
end proc;

procedure process-end-confirmed
    waiting-for := waiting-for −1;
    if waiting-for = 0 and idleness > 1 then
        if BFST-leader then
            send-end(parent)
        else
            send-end-confirmed(BFST-parent)
        end if
    end if;
    if waiting-for = 0 and idleness ≤ 1 then
        process-end-negative
    end if;
end proc;

procedure process-tuple
    idleness := 0;
    ...
end proc;
```

Figure 2: Protocol for distributed termination of a request, with *request-id* as implicit parameter

4 The Monotone Flow Property

Intuitively we may think of information passing as function evaluations in which "c" and "d" arguments are inputs and "f" arguments are outputs. Frequently the input "d" bindings flow through the subgoals and return to bind "f"s in the goal. The definition of the *monotone flow property* is intended to cover the class of rules having a natural, efficient way for the bindings to flow, and is closely analogous to Sagiv's uniqueness condition for database schemes [Sag83]. I am indebted to Y. Sagiv for numerous helpful discussions on the material in this section. It is based on the notion of hypergraph acyclity. Recall that a hypergraph is a generalization of a graph in which hyperedges are arbitrary sets of nodes instead of just pairs of nodes.

4.1 Evaluation Hypergraphs and Qual Trees

Definition 4.1: Given a rule in which each argument in the head of the rule has a binding status, its *evaluation hypergraph* is the hypergraph with a node for each variable in the rule, and hyperedges as follows:

- The variables in the head of the rule that have "c" and "d" binding classifications comprise the hyperedge of the head. We use a superscript b to underscore the fact that this hyperedge contains only bound variables.
- The hyperedge of each subgoal consists of all variables in that subgoal.

□

The intuition behind this definition is that evaluating the rule for the bindings in the head can be viewed as evaluating a join expression in which the bindings in the head are one relation and the subgoals are the remaining relations.

Definition 4.2: We say that a rule (with the given binding classifications) has the *monotone flow property* if its evaluation hypergraph is acyclic in the sense of [BFM*81,Yan81]. This version of acyclicity is also called α-acyclicity. □

Example 4.1: Consider the following Prolog-style rules, where the goal p is on the left and the subgoals are on the right. (Read "←" as "if.")

R1: $p(X,Z) \leftarrow a(X,Y),\ b(Y,U),\ c(U,Z).$

R2: $p(X,Z) \leftarrow a(X,Y,V),\ b(Y,U),\ c(V,T),\ d(T),\ e(U,Z).$

R3: $p(X,Z) \leftarrow a(X,Y,V),\ b(Y,W,U),\ c(V,W,T),\ d(T),\ e(U,Z).$

Assume the first argument of p is a "d" and the second is an "f". In the first rule, we see that information "flows" from X to Y to U to Z quite naturally. In the second rule, information flows from X to both Y and V. Now we have a choice of whether to extend the flow next to U by evaluating subgoal b, or to T by evaluating c. But it really doesn't matter, as these two extensions are independent; in fact, they can be done in parallel. The first two rules both have the monotone flow property. For example, the hypergraph of rule R2, shown in Fig. 3, is clearly acyclic.

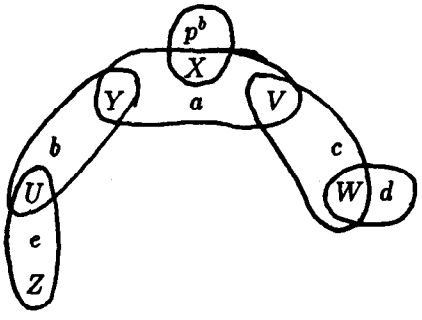


Figure 3: Hypergraph for rule R2.

However, in the third rule, in which the flow begins the same way as in the second, the choice of where to go after a *does* matter. If we evaluate b next, we get W bindings to help restrict c, and *vice versa*. On the other hand, if we do both in parallel, we risk computing two large relations that are nearly unjoinable due to mismatches on W. The third rule fails to have the monotone flow property, because of a cycle involving Y, V, and W, as shown in Fig. 4. □

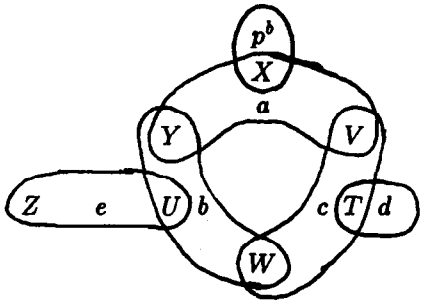


Figure 4: Hypergraph for rule R3.

Acyclic hypergraphs have numerous nice properties [BFM*81,Yan81]. The first one we shall make use of is the fact that the *Graham reduction procedure* both tests for acyclicity and exhibits a *qual tree* for the hypergraph when it is acyclic. The qual tree is an undirected acyclic graph whose nodes are the head and subgoals of the rule (i.e., the hyperedges of the hypergraph). (The remainder of the definition is given later.) The edges of the qual tree are output by the procedure, which consists of applying these two reductions to the hypergraph as long as possible:

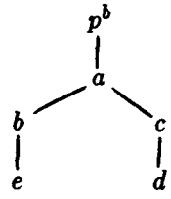
1. If a variable is currently in only one hyperedge, delete it.
2. If a hyperedge h_1 is a subset of another hyperedge h_2, add an edge between h_1 and h_2 to the qual tree and delete h_1 from the hypergraph.

It is known that a hypergraph is acyclic if and only if this procedure reduces it to one empty edge. In this case a qual tree has been constructed. We choose the node corresponding to the head of the rule as the root.

The important *qual tree property* that makes a tree a qual tree is the following: For any variable in the rule, and any two hyperedges (rule head or subgoals) containing that variable, the path between those hyperedges in the qual tree only involves hyperedges (qual tree nodes) that also contain that variable. Cyclic hypergraphs do not have qual trees, but have qual graphs containing (undirected) cycles.

Example 4.2: The qual tree for rule R2 of Example 4.1 with bindings $p(X^d, Z^f)$, whose hypergraph

was shown in Fig 3 is



By directing all edges away from the root, we obtain the greedy information passing strategy that goes with this qual tree It is the one described in Example 4 1 □

Theorem 4.1: The information passing strategy obtained by directing all qual tree edges away from the root (i e , the qual tree node for the rule head) is a greedy one

Proof We build the graph that will define the information passing strategy by copying the qual tree We show by induction on the number of nodes already added to the information passing graph that each node has maximum bound variables when it is added to the graph It is clearly true for the root, with 0 nodes in the graph, because it is the only node that initially has bindings For $k > 0$, we define the k-adjacency to be the set of nodes not yet in the information passing graph, but adjacent (in the qual tree) to a node already in the graph Select as the k-th node to add a node a that has maximum bindings among nodes in the k-adjacency We claim that if c is not in the k-adjacency (and not in the graph), then c has no more bindings than a Let b be the qual tree ancestor of c that is in the k-adjacency By the qual tree property, every bound variable in c also appears in b, since bindings propagate from nodes already in the graph But b has no more bindings than a, proving the claim ▮

4.2 Extendable Monotone Flow

Consider an information passing rule/goal graph containing two rule nodes v and w, such that a path from w to v passes thru only goal nodes This implies that the head of the rule at w corresponds to a subgoal, say p, in the rule at v, and their argument bindings also match Possibly v and w are the same node, but we consider separate copies of their rules, which we call R_v and R_w Assume that R_v and R_w have the monotone flow property, and that their information passing strategies correspond to the qual trees of their evaluation hypergraphs, as discussed in the previous paragraphs

Suppose we "extend" R_v by resolving on p with R_w, in other words, first unify the head of R_w with subgoal p, then replace p in R_v by the subgoals of R_w Let the argument bindings for the head of the extended rule be the same as R_v Can we say anything about this extended rule, in regards to the *monotone flow property*? The question is especially interesting when the extension is done by resolving on a recursive subgoal, because then the property might be transmitted to all recursive extensions of the rule

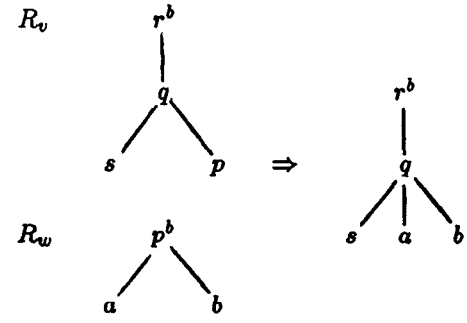


Figure 5 Qual tree composition

Theorem 4.2: Let v, w and subgoal p be as described above Suppose p appears as a leaf in the qual tree of v Then the qual trees of v and w compose to produce a qual tree for the extended rule produced by resolving R_v and R_w on p The composition of qual trees is performed as follows Attach the *neighbors* of the root p^b of the qual tree of w to the *parent* of the resolved leaf p in the qual tree of v, removing both p^b and p (See Fig 5, the analogy to resolution is clear)

Proof It is clear that the composed tree has a node for each subgoal of the extended rule, and has a node for the bound variables of the head Recall that qual tree nodes are actually hyperedges of variables It remains to show that the composed tree has the qual tree property, namely that if any variable X appears in two distinct nodes of the composed tree, it also appears in every node on the path between them The only problem is when the two nodes were in different qual trees before the composition But the variables of the two rules are disjoint except for the effects of unification, and all variables affected by unification appear in subgoal p of R_v and/or the head of R_w Since p is a leaf in the qual tree, it is a leaf in the information passing graph as well, which implies that the free variables of p appear nowhere else in the qual tree of R_v Moreover, all bound variables of p appear in the qual tree parent of p (by the qual tree

property) It follows that if variable X appears in two nodes, say s from the qual tree of R_v and b from the qual tree of R_w, then X appears in the parent of p, and also in p^b (which is not in the composed tree) But we know then that X appears everywhere in the path from b to some neighbor of p^b, and that neighbor is now adjacent to the parent of p in the composed tree Finally, X appears everywhere in the path from the parent of p to s, verifying the qual tree property ∎

4.3 Monotone Flow and Efficiency

We conjecture that under "reasonable assumptions" the optimal information passing strategy for rules having the monotone flow property is the greedy one suggested by the above examples, and based on a qual tree Our "reasonable assumptions," which assert a high degree of ignorance about the relations in the EDB, are

- The relations of all subgoals are of comparable size, and large
- Each bound argument reduces the relation size by an order of magnitude,[5] with a corresponding reduction in retrieval cost (Bound arguments function as selections)
- The size of a join relation is the size of the cross product, reduced by one order of magnitude for each pair of join arguments, i e , each pair of subgoal arguments containing the same variable
- The cost of computing a join is proportional to the sum of the sizes of the operands and the size of the result
- Multiplicative log factors, such as would arise from sorting or maintaining indices, can be ignored, either because some hashing method is used, or because they are about the same for all alternatives

Our conjecture is based on the algorithm in [Yan81] for computing joins over acyclic schemes That algorithm uses the qual tree and works essentially in two stages In the first stage, a series of semijoins analogous to our information passing is carried out to prune the relations (nodes in the qual tree) down to pairwise consistency In the second stage, the pruned relations are joined using the qual tree as an expression tree The acyclicity and pairwise consistency guarantee that the temporary relations formed in the second stage grow monotonically, hence their size is bounded by the size of the final result Because our computation involves recursive subgoals in general, we cannot exactly imitate this algorithm Nevertheless, the greedy information passing strategy, supported by qual trees in the case of monotone flow rules, seems to be a well motivated heuristic at the very least

[5] We say n is "reduced by an order of magnitude" if its logarithm is reduced by some constant factor $\alpha < 1$, where the same α is used for all size and cost estimates that arise E g , if $\alpha = \frac{1}{2}$ is used, and the original relation has size n, we assume that selection on one argument yields about $n^{\frac{1}{2}}$ tuples and selection on two arguments yields about $n^{\frac{1}{4}}$ tuples

References

[AU79] A V Aho and J D Ullman Universality of data retrieval languages In *6th ACM Symp on Principles of Programming Languages*, pages 110–117, 1979

[BFM*81] C Beeri, R Fagin, D Maier, A Mendelzon, J D Ullman, and M Yannakakis Properties of acyclic database schemas In *13th STOC*, pages 355–362, 1981

[HN84] L J Henschen and S A Naqvi On compiling queries in first-order databases *JACM*, 31(1) 47–85, 1984

[Loz85] E L Lozinskii Evaluating queries in deductive databases by generating In *Proc 9th Int Joint Conf on Artificial Intelligence*, pages 173–177, 1985

[MS81] D McKay and S Shapiro Using active connection graphs for reasoning with recursive rules In *Proc 7th Int Joint Conf on Artificial Intelligence*, pages 368–374, 1981

[Por85] H H Porter, III *Earley Deduction* Technical Report , Oregon Graduate Center, Beaverton, OR, 1985

[Rei78] R Reiter On closed world databases In Gallaire and Minker, editors, *Logic and Databases*, pages 55–76, Plenum Press, New York, 1978

[Sag83] Y Sagiv A characterization of globally consistent databases and their correct access paths *ACM Trans on Database Systems*, 8(2) 266–286, June 1983

[Ull82] J D Ullman *Principles of Database Systems* Computer Science press, Rockville, Md , 1982

[Ull84] J D Ullman *Implementation of logical query languages for databases* Technical Report STAN–CS–84–1000, Dept of Computer Science, Stanford University, Stanford, CA, May 1984

[Var82] M Y Vardi Complexity of relational queries In *14th ACM Symposium on Theory of Computing*, pages 137–145, 1982

[VEK76] M H Van Emden and R A Kowalski The semantics of predicate logic as a programming language *JACM*, 23(4) 733–742, 1976

[Vie85] L Vieille *Recursive axioms in deductive databases various solutions* Technical Report KB-6, European Computer-Industry Research Center, Munich, Germany, 1985

[Wal81] A Walker *Syllog A Knowledge Based Data Management System* Technical Report 034, Dept of Computer Science, New York University, 1981

[Yan81] M Yannakakis Algorithms for acyclic database schemes In *Proc 7th Int Conf on Very Large Databases*, pages 82–94, Cannes, France, Sep 1981