A Framework for an Efficient Implementation of Deductive Databases*

Michael Kifer
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794, USA

Eliezer L. Lozinskii
The Hebrew University
Department of Computer Science
Jerusalem 91904, Israel

August 27, 2005

Abstract

We describe a method for query evaluation in deductive databases which is based on *dynamic filtering* of data flow. The basic query evaluation strategy is bottom-up and set-oriented. The method imposes no restrictions on the form of Horn axioms, takes advantage of actually stored data, allows compile time preprocessing, and is well suited for parallel and distributed execution.

1 Introduction

Query evaluation in deductive databases may be very much time consuming, which is due to exponential complexity of the resolution principle that provides basis to almost all known deductive systems. The two main search directions in such systems are $top - down$ and $bottom - up$. Being utilized in a straightforward and unoptimized manner, both strategies may be terribly inefficient. Although an efficient evaluation of a given query may require a relatively small amount of work, an unoptimized deduction process tends to perform lots of unproductive derivations which do not contribute to the query. Thus, for an efficient implementation of a deductive database system, a number of questions have to be positively answered, and the corresponding problems solved:

(1) What are the facts necessary for a given query evaluation? Can we perform query evaluation in such a way that (almost) only relevant facts will be generated?

Neither of the two strategies (top-down or bottom-up) is always the best, and there are well known cases in favour of each of them; each one of these approaches has its inherent shortcomings and advantages. While the top-down derivation resides on the syntactic structure of axioms, the bottom-up generation is controlled by the actual data stored in the database. The latter feature is a strong advantage of the bottom-up approach from the point of view of system performance, because, to a large extent, what makes a query easy or difficult in a given database is its current set of facts. For instance, set of facts may determine just how many loops of a recursive rule will be actually executed for a given query evaluation. Thus, an important question is

(2) How to devise an optimized query evaluation method that combines positive features of both top-down and bottom-up approaches, and avoids (most of) their shortcomings?

*Research of M. Kifer was supported in part by the NSF grant DCR-8603676. The work of E.L. Lozinskii was supported in part by the Israel National Council for Research and Development under the grant 2454-85, and by the NSF grant DCR-8603676; it was partially performed at SUNY at Stony Brook while on a sabbatical leave from the Hebrew University.

An important issue in deductive systems is a measure to which query evaluation programs are precompiled. If most of the processing is left to run time, then query evaluation might be too time consuming because of the overhead usually associated with control. On the other hand, if a strategy is "too precompiled" then inefficiency arises because the actually stored data is ignored to a large extent. Hence, such systems tend to overlook possible optimizations that dynamic systems are more likely to find. Thus, we have the following problem:

(3) Which characteristics of a given database system are common to all (or most) queries such that their role in query answering can be predicted and preprocessed at a reasonable expense of time and space? Such partial preprocessing will relieve the run time query evaluation from unnecessary search. The second aspect of this question is, how specific parameters of a given query can be exploited dynamically at run time in order to perform the evaluation efficiently and significantly reduce the amount of redundant derivations?

The interest in deductive databases [5] emerged from the success of Logic Programming, of which PROLOG is the most popular tool. However, along with all its power and popularity, PROLOG has certain features unacceptable for databases (like the tuple-at-a-time processing, logical incompleteness, etc.). Recently a number of elegant approaches to deductive databases have been proposed [14, 7, 17, 1, 8, 15, 6] A brief comparison with some of these methods is given in Section 9. Due to the heuristic nature of deductive query evaluation methods their limitations look not at all surprising. The more restricted is a method the more efficient it could be in its favourite subfield of problems. In this paper we describe a method general enough to be applicable to all Horn function free deductive databases (sometimes called $DATALOG$), and well competitive with specialized methods.

2 Deductive Databases: Background

A *deductive database* consists of a set of facts (ground atoms) called *extensional database* (EDB) and a set of logical axioms (first order formulas) called *intentional database* (IDB). Facts are organized in relations (represented by predicate names), and axioms are universally quantified function-free Horn clauses. Relations can be represented *extensionally*, i.e. by a set of facts, *intentionally*, i.e. implied by axioms, or in both ways. *Base relations* are allowed to have an extensional component, i.e. some of the tuples may be explicitly stored in the database while the others can be derived using the axioms. Nonbase relations are assumed to be purely intentional.

3 Representation of Axioms

We shall use a special notation for axioms, called *AC-notation*, which is useful in describing data flow during query evaluation. A vector of *distinct* variables is associated with each relation name so that different predicates do not share variables. These variables are related to each other by a condition attached to the axiom (which suggests the name AC-notation). Thus, axioms will be represented as follows:

$$\alpha: \; R(\bar{r}) \leftarrow Q_1(\bar{q_1}/1), \ldots, Q_n(\bar{q_n}/n), Cond_\alpha, \tag{1}$$

(i) where α is the axiom name;

(ii) $\bar{r}$, $\bar{q_1}$, ... ,$\bar{q_n}$ are the vectors of variables associated with R, Q_1, ... , Q_n, respectively (the Q_i need not be distinct). A predicate name can appear several times in an axiom α, so we use $\bar{\mathbf{p}}/j$ to denote the vector of variables associated with predicate P occupying the j-th position within the body of α;

(*iii*) $Cond_\alpha$ is the *attached condition* of α of the form $(u1\ \theta_1\ v1) \wedge (u2\ \theta_2\ v2) \wedge \ldots$, where $u1$, $v1$, $u2$, $v2$, ..., are variables appearing in $\bar{r}$, $\bar{q_1}/1$, ..., $\bar{q_n}/n$, and θ_i's stand for elementary comparisons $=$, $>$, $<$, $>=$, $<=$.[1]

For instance, axiom $\beta: A(x,y) \leftarrow A(x,z), B(z,w)$, .br $A(w,y)$ will be represented as

$$\beta: \quad A(a1\,,\,a2) \quad \leftarrow \quad \begin{array}{l} A(a1/1\,,\,a2/1),\ B(b1/2\,,\,b2/2),\ A(a1/3\,,\,a2/3), \\ a1 = a1/1,\ a2/1 = b1/2,\ b2/2 = a1/3,\ a2 = a2/3 \end{array}$$

It should be emphasized that the AC-notation is just a convenient formalism. The user still writes the rules in a conventional notation, and conversion to the AC-notation is done automatically.

4 System Graphs and Queries

Let $\mathbf{S}$ be a collection of axioms. *System graph for* $\mathbf{S}$ [13], denoted $SG(\mathbf{S})$, is a directed bipartite AND/OR graph whose node set consists of *rel-nodes* and *ax-nodes*: exactly one rel-node (resp., ax-node) corresponds to each relation name (resp., axiom) appearing in $\mathbf{S}$. In Figure 1 rel-nodes are depicted as rectangles, while ax-nodes are displayed as ovals enclosing the corresponding attached conditions. Let α be an axiom of the form $R(\bar{r}) \leftarrow Q_1(\bar{q_1}/1), ..., Q_n(\bar{q_n}/n), Cond_\alpha$. Then $SG(\mathbf{S})$ contains arcs going from ax-node α to rel-node R, and from each one of rel-nodes Q_1, ..., Q_n to ax-node α. The arc from Q_i to α is labeled with i_α.

***Example* 4.1** The following set of axioms defines the transitive closure of relation $P(p1, p2)$ selected on the second argument, and projected on the first (see Figure 1).

$\alpha: A(a1, a2) \leftarrow P(p1/1, p2/1), a1 = p1/1, a2 = p2/1$
$\beta: A(a1, a2) \leftarrow P(p1/1, p2/1), A(a1/2, a2/2),\ a1 = p1/1, p2/1 = a1/2, a2 = a2/2$
$\gamma: \mathbf{Ans}(ans1) \leftarrow A(a1/1, a2/1),\ ans1 = a1/1,\ a2/1 = e.$

Without loss of generality we will only consider queries of the form $\{\bar{x}|\mathbf{Ans_q}(\bar{x})\}$, where $\mathbf{Ans_q}$ is a special *query predicate* (one per each query $\mathbf{q}$) defined by the following set $\mathbf{S'}$ of axioms (represented in the conventional notation):

$$\mathbf{S'} \quad \left\{ \begin{array}{lll} \mathbf{Ans_q}\,(\bar{x}) & \leftarrow & Q_{11}\,(\bar{x}\,,\,\bar{y}),\ ...,\ Q_{1n_1}\,(\bar{x}\,,\,\bar{y}) \\ \ldots & \ldots & \ldots \\ \mathbf{Ans_q}\,(\bar{x}) & \leftarrow & Q_{k1}\,(\bar{x}\,,\,\bar{y}),\ ...,\ Q_{kn_k}\,(\bar{x}\,,\,\bar{y}) \end{array} \right.$$

where Q_{ij} are some database predicates. Let $\mathbf{S}$ denote the set of axioms of the intentional database. Then $SG(\mathbf{S} \cup \mathbf{S'})$ is called the *system graph for query* $\mathbf{q}$.

With this definition, one can view the third axiom of Example 4.1 as a query to a database whose intentional part is given by the first pair of axiom. The graph of Figure 1 will then be the SG for that query, while the lower part of the graph corresponds to the SG of the intentional database.

5 Data Flow in System Graphs and Query Evaluation

System graphs are used in our approach as a tool for query evaluation representing flow of data among the rel- and ax-nodes. With this intention in mind the arcs will be usually referred to as *ports*. An ax-node has exactly one *output port* and one or more *input ports*. A rel-node may have zero or more input and output ports. Rel-nodes without input ports are base relations.

Data is sent along the ports only in the directions indicated by the arrows. For instance, rel-node $P(p1, p2)$ in Figure 1 may send a tuple, $< b, e >$, to ax-node α along the port 1_α. Variables $p1/1_\alpha$ and $p2/1_\alpha$ of α will receive the values of variables $p1$ and $p2$ of $P(p1, p2)$, respectively. Node

[1] For simplicity, in this paper we consider equalities only.

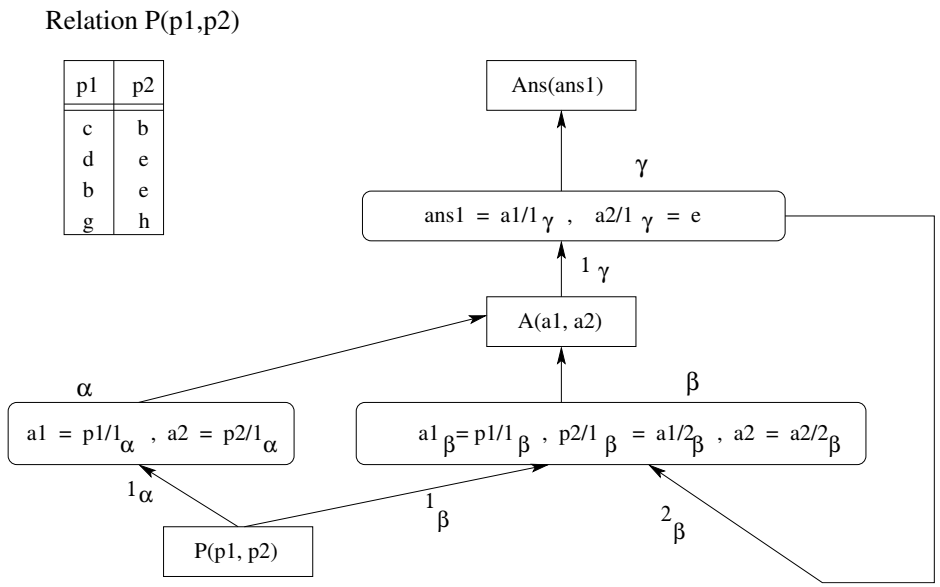

Figure 1: A System Graph

α can further send the tuple $< b, e >$ to the rel-node $A(a1, a2)$, so that $a1$ will receive the value b of $p1/1_\alpha$, and $a2$ will get the value e of $p2/1_\alpha$, as indicated by the equations in α.

Next, $A(a1, a2)$ can send $< b, e >$ to β along the port 2_β. In β the variables $a1/2_\beta$ and $a2/2_\beta$, associated with port 2_β, will get the values of $a1$ and $a2$, respectively. At that point $P(p1, p2)$ may have sent tuple $< c, b >$ to β via 1_β, which assigns value c to $p1/1_\beta$ and b to $p2/1_\beta$. The two tuples satisfy the attached condition of β, so $< c, e >$ is generated from $< c,\ b >$ and $< b,\ e >$, and is sent to $A(a1, a2)$, etc.

"Slashed" variables $p1/1_\alpha$, $p2/1_\alpha$, $a1/2_\beta$ etc. are *input variables* receiving their values via input ports from the corresponding variables associated with the preceding rel-nodes. For instance, $p1/1_\alpha$ gets its values via port 1_α from variable $p1$ of $P(p1, p2)$. "Non-slashed" variables of axioms (e.g., $a1$, $a2$ of α) are *output variables*. When the attached condition of an axiom, α, is satisfied then its output variables are instantiated and the tuple of output values is sent out via the single output port of α. For simplicity we assume that variables of the head predicate of an axiom also appear in its body. That is, in the ax-nodes each output variable is bound to an input variable.

Now we can think of the following *simplistic query evaluation*. Computation starts when all the base rel-nodes send out all their tuples (or appropriate projections thereof) via all their *relevant* output ports. A port is *relevant* to a query, **q**, if it is a part of a directed path to the query node $\mathbf{Ans_q}$. Ax-nodes store tuples that they receive from each input port in a local file associated with this port. As soon as an ax-node discovers that some sets of tuples in its local files can be joined (more precisely, θ-joined, where $\bar{u}\bar{\theta}\bar{v}$ is the attached condition of the axiom) to produce *new* output tuples that were not produced before, it performs the join and sends a file of new tuples through the output port. Rel-nodes accumulate all tuples they receive. When a set of tuples arrives at a rel-node the tuples are checked against the information stored at that rel-node. New tuples that are not yet stored are saved, and then sent out via all relevant output ports of that rel-node. Computation terminates when no more new tuples can be generated by the system. Distributed termination of this process is based on the algorithm described in [13]. It can be shown [18] that the simplistic evaluation is sound and complete.

It should be emphasized that data flow in SGs is actually set-oriented (as opposed to tuple-at-a-time approaches, such as PROLOG). Rel-nodes actually send all their new tuples as a file,

and ax-nodes can join files in any suitable way (nested loops, sort-merge, etc.), depending on characteristics of those files.

6 Data Flow Optimization: a Motivation

It can be noticed that the simplistic query evaluation allows transmission along the ports of some facts that are useless for answering the query. This, in turn, may cause generating of other useless facts, etc. Thus a natural optimization of the simplistic query processing would be filtering out as many useless facts as possible. One way of achieving this goal would be restricting the ports with special conditions, $filters$, which let certain facts through and block useless ones up.

Coming back to Example 4.1 we see that one may push selection, $a2/1_{\gamma=e}$, found in the attached condition of γ downward, which results in imposing filters $p2 = e$ on port 1_α, and $a2 = e$ on both ports 1_γ and 2_β (filters are not shown in Figure 1). The effect of these filters is that certain tuples will not pass through the ports, which spells a more economical computation. A general way of computing such filters has been recently described in [11].

This type of filters is called *static*: they are computed at compile time, and do not take into consideration actual facts stored in the database, which determines the limitations of static filtering.

***Example* 6.1** Consider the data for relation $P(p1, p2)$ shown in Figure 1. By direct inspection we see that imposing of static filters $p2 = e$ and $a2 = e$ on 1_α and 2_β, respectively, does some useful work by blocking tuples $< c, b >$ and $< g, h >$ up from passing through these ports. However, $< b, e >$ and $< d, e >$ will come through port 1_β, although this does not contribute to the answer.

Static filters (see [11] for more details) correspond to what is known in the query optimization theory as *algebraic manipulations* with relational expressions [16], therefore the limitations of static filtering are not at all surprising. Performance of filters can be certainly improved if they are dynamically formed in accordance with the actual data utilized in the course of query evaluation. This can be achieved only at query run time, so we call the approach described in the next sections *dynamic filtering*. Dynamic filtering generalizes what is known as the *query decomposition* approach [19].

7 Query Optimization by Dynamic Filtering

7.1 An Illustration

With dynamic filtering query evaluation starts with assigning *initial values* to the filters on ports relevant to the query. This is done at query compile time. If the initial filters do not let any tuple to pass through then the computation terminates with the vacuous answer. Otherwise, sets of tuples selected due to the initial filters start traveling along the ports causing changes in other filters (as described farther in this paper), which lets other tuples to pass through, and so on, until all answers to the query are produced.

***Example* 7.1** To see dynamic filters at work let us apply this idea to Examples 4.1 and 6.1. At the beginning initial filters $p2 \in \{e\}$, $a2 \in \{e\}$, and $a2 \in \{e\}$ are placed on ports 1_α, 2_β and 1_γ, respectively. Other ports are initialized to **false** (blocked). The choice of initial filters is suggested by the selection $a2/1_\gamma = e$ specified in the query ax-node γ of Figure 1. The initial filter on 1_α selects a set of tuples, $\{< d, e >, < b, e >\}$, and then lets it through the port 1_α. Next, this set of tuples proceeds to node $A(a1, a2)$ and then goes simultaneously through ports 1_γ and 2_β. Ax-node γ generates and sends out its first partial set of answers, $\{< d >, < b >\}$.

A more interesting processing takes place in node β. To generate a new fact, β needs tuples from both its input ports, 1_β and 2_β. Some tuples ($< d, e >$ and $< b, e >$) have arrived at port 2_β, but 1_β is still closed. To use these tuples β needs matching tuples from 1_β, so it places filter

$p2 \in \{d, b\}$ on this port. This is considered as a *sideways propagation* of data within *beta*. As a result, $\{< c, b >\}$ is selected by the filter and passed via 1_β from the rel-node $P(p1, p2)$ to β. Now β can join its local files and generate tuple $< c, e >$, which is then sent to the rel-node $A(a1, a2)$. This eventually leads to another answer, $< c >$, to the query. Node $A(a1, a2)$ also sends $\{< c, e >\}$ via 2_β to β which, in turn, updates the filter on 1_β from $p2 \in \{d, b\}$ to $p2 \in \{d, b, c\}$ (by the sideways propagation).

In addition, β tries to make more use of its acquisition, tuple $< c, b >$, arrived previously at 1_β. So, it adds a disjunct $a1 \in \{b\}$ to the already existing filter $a2 \in \{e\}$ on port 2_β, updating it (again, by the sideways propagation) to $a1 \in \{b\} \vee a2 \in \{e\}$. As soon as a filter is changed, the change must be "pushed back" into the system graph. This "push-back" is determined by the equations in ax-nodes and is pretty similar to filter initialization. This action is considered as *backward propagation* of data. As a result of the previous filter update, the filter on port 1_α is changed from $p2 \in \{e\}$ to $p1 \in \{b\} \vee p2 \in \{e\}$, and the filter on 1_β changes from $p2 \in \{d, b, c\}$ to $p1 \in \{b\} \vee p2 \in \{d, b, c\}$. Tuple $< b, e >$ now passes from $P(p1, p2)$ to β via 1_β, updating the filter on 2_β from $a1 \in \{b\} \vee a2 \in \{e\}$ to $a1 \in \{b, e\} \vee a2 \in \{e\}$ (by sideways propagation). This in turn changes (by backward propagation) the filter on 1_α from $p1 \in \{b\} \vee p2 \in \{e\}$ to $p1 \in \{b, e\} \vee p2 \in \{e\}$, and the filter on 1_β from $p1 \in \{b\} \vee p2 \in \{d, b, c\}$ to $p1 \in \{b, e\} \vee p2 \in \{d, b, c\}$. But this time no new tuple arrives at any of the ax-nodes, hence no new tuple is generated and we end up with the answer $\{< d >, < b >, < c >\}$.

Observe that in Example 7.1 useless tuples are blocked up. Indeed, $< c, b >$ and $< g, h >$ were not sent to the ax-node α (as in static filtering), and $< d, e >$ did not get through to β (in contrast to static filtering). In general, it can be shown that neither strategy (dynamic or static) supersedes the other one, and both can be used cooperatively. Due to this combined method each filter becomes a conjunction of the static part (determined at compile time) and of the dynamic part (computed and updated at run-time). Since static filters are described in details in [11], we will not consider them here.

It should be noted that filters select *sets of tuples*, and send them to the appropriate ax-nodes as a single package. As in the simplistic evaluations, ax-nodes can take advantage of the most suitable join strategy, depending on characteristics of the files being joined.

7.2 Computing Dynamic Filters

From now on we view filters as dynamic conditions controlled by changes occurring during query evaluation. Typically filters are updated when sideways or backward propagation of data takes place, as explained in Section 7.1. Obviously not all filters are affected by any particular propagation (e.g., when $a1 \in \{b\}$ was added to the filter on port 2_β, then backward propagation of this change has affected the filter on 1_α, but not on 1_γ). It turns out that for each filter, F, in an SG, the set of filters affected by any change to F can be precomputed at the system compile time.

For that purpose we have to determine *data sources* for each occurrence of a variable. This concept was inspired by *migration sets* [12]. Informally, data source set for a variable x occurring in a rel-node, R, consists of all those variables from which x can receive values during the simplistic query evaluation.

***Definition* 7.2** Let $R(..., x, ...)$ be a rel-node and x be one of its variables. Then *data source set* for x is a set, denoted $SRC(x)$, of certain input variables appearing in ax-nodes, that are predecessors of R in the SG. $SRC(x)$ is recursively defined as follows:

proc *data-source-of*(x)
  global var $SRC := \emptyset$;
  add-data-source-of(x)
endproc /* data-source-of */

proc *add-data-source-of*(x)
  for each *immediate predecessor ax-node* α *of* R **do**
    $SRC := SRC \cup \{y/i_\alpha | Cond_\alpha \; implies x = y/i_\alpha\}$
    /* recall: y/i_α is an input variable of α */

    for each *new member* y/i_α *of* SRC **do**
      add-data-source-of(y)
      /* y is a variable of a rel-node, S, connected
        to α via i_α; y/i_α gets its values
        from y along the port i_α */
    endfor
  endfor
endproc /* add-data-source-of */

***Example* 7.3** Here are some sources of variables for the SG of Figure 1.

$SRC(ans1) = \{a1/1_\gamma\,,\; p1/1_\alpha\,,\; p1/1_\beta\}$
$SRC(a1) = \{p1/1_\alpha\,,\; p1/1_\beta\}$
$SRC(a2) = \{p2/1_\alpha\,,\; a2/2_\beta\}$

We will now define sideways and backward propagations formally. In this discussion we adopt the following form of filters: Let i_α connect a rel-node $R(r_1, ..., r_k)$, with an ax-node α. A filter on i_α is a condition of the form $\vee_i\, r_i \in Range(r_i)$, where $Range(r_i)$ is a subset of the domain of r_i (in particular, the entire domain or an empty set).

Consider a filter $Fbar$ on an output port of a rel-node $R(r_1, ..., r_k)$, and suppose $Fbar$ is changed so that $Range(r_i)$ is augmented by a set of values Δ, i.e., from now on $Fbar$ lets also tuples with $r_i \in \Delta$ through. But r_i gets its values only from its sources, hence the filters controlling ports occurring in $SRC(r_i)$ must also be modified to allow the appropriate tuples to pass through. This is called a *backward propagation* of data. Observe that data sources are defined in such a way that those and only those filters have to be updated which control the ports occurring in $SRC(r_i)$. Formally, let $p/j_\alpha \in SRC(r_i)$ and suppose F_{j_α} is a current state of the filter on j_α. Then the backward propagation of the update $r_i \in \Delta$ (made to the aforesaid filter $Fbar$) changes the state of the filter on j_α to $F_{j_\alpha} \vee p \in \Delta$ (recall that p is the variable of the rel-node, say $P(..., p, ...)$, from where p/j_α gets its data via port j_α).

As it has been informally described in Section 7.1, query evaluation begins with filter initialization. This can be done by a backward propagation of bindings specified in the query, as illustrated in Example 7.1.[2] The initial filters select appropriate sets of tuples from the base relations, and let them through to the appropriate ax-nodes. When a bundle of tuples reaches an ax-node, α, certain input variables of that node become instantiated. For instance, in Example 6.1, when $\{< d, e >, < b, e >\}$ arrives at β via port 2_β, then $a1/2_\beta$ and $a2/2_\beta$ get sets of values $\{d, b\}$ and $\{e\}$ respectively. These values propagate to other input variables within the ax-node according to the attached condition, so $\{d, b\}$ is assigned to $p2/1_\beta$. To generate output tuples β needs matching tuples from 1_β. To let the matching tuples through 1_β, the corresponding filter must be updated. Thus, if there are two input variables, x/i_α and y/j_α, of an ax-node α such that $Cond_\alpha$ implies $x/i_\alpha = y/j_\alpha$, whenever x_α is assigned a new set Δ of values, then Δ is propagated *sideways* to the filter controlling j_α, updating it from F_{j_α} to $F_{j_\alpha} \vee y \in \Delta$. Whenever sideways propagation changes a filter, the change is subsequently spread in the SG by the backward propagation as described earlier.

[2] Actually, there are certain subtleties arising when no bindings are specified, or when bindings are also specified in axioms other then the axioms defining the query rel-node **Ans**. These details are not essential in order to understand the method, and will be taken care of elsewhere.

7.3 Further Improvements: Sideways Propagation Graphs

Dynamic filtering, as described above, suggests certain further improvements. Indeed, this method is not *useless free* in the sense that useless facts may be sent and generated in the course of query evaluation. Perhaps, useless freedom in general cannot be captured by a universal algorithm, however, intuitively this is a sound goal justifying efforts towards reducing the amount of useless data. One direction is: instead of relatively "loose" filters of the form $\vee_i r_i \in Range(r_i)$ of Section 7.2, use more "precise" filters of the form $\vee_j(\wedge_i r_i \in Range_j(r_i))$. Unfortunately, space and time overhead associated with precise filters seems to be high, and their efficient implementation requires more research.

Next, we observe that uncontrolled sideways propagation may open filters "too widely", causing too eager a query evaluation, allowing some useless tuples to slip through. A remedy may come from restricting the directions in which sideways propagation is allowed (this is similar to techniques used for parallel execution of PROLOG programs [4, 3]. For that purpose we install in each ax-node, α, a directed *sideways propagation graph* (abbr. SPG), whose vertexes are input ports of α, and directed arcs indicate the directions in which sideways propagations is allowed. Namely, an input port i_α may expect matching data from a sibling port j_α only if the SPG of α has an arc going from i_α to j_α.

For instance, in Example 6.1 a good SPG for β consists of a single arc from 2_β to 1_β. With this SPG, new tuples arriving at port 1_β will not cause sideways propagation, which may significantly improve the performance. Indeed, suppose that relation $P(p1, p2)$ in Example 6.1 has tuple $< b,\ f >$ in addition to the tuples it had before. Then, in the course of query evaluation described in Example 7.1, the tuple $< c,\ b >$ arriving at port 1_β should cause sideways propagation of $a1 \in \{b\}$ followed by the backward propagation, which eventually brings the useless tuple $< b,\ f >$ at ports 1_α and 1_β. However, if sideways propagation is restricted by the above SPG, this does not happen. Thus, SPGs can reduce both data flow and overhead associated with managing of dynamic filters.

Constructing of an efficient SPG in general is a difficult problem. Inaccurately chosen SPG may even destroy completeness of the system. Space limitations do not allow us to extend more about SPGs, but to say that one general form of SPG that renders completeness to a database is the *query graph* [9], which is a special case of the *connection graph* [16] (constructed from the attached conditions of ax-nodes).[3] In certain cases an SPG, which is better than just the query graph, can be constructed. Indeed, the sideways propagation scheme of Example 7.1 corresponds exactly to an SPG built from a query graph, while the SPG constructed in the previous paragraph has been shown to be more efficient.

The ability to choose different SPGs gives our approach flexibility, which is akin to *capture rules* [17]. Various special cases (like linear axioms [1]) can be efficiently solved by "customizing" SPGs. Furthermore, suppose we already have SPGs that make query evaluation for a particular SG complete. It can be shown that adding of new directed arcs to the SPGs will not impair completeness. However, additional arcs will make query evaluation more eager. Indeed, these arcs cause more sideways propagations, which tend to make filters weaker. Hence certain tuples may pass earlier through the ports, and the answer to the query may be generated earlier too. On the other hand, the eager evaluation may let more useless tuples through, hence one can trade eagerness for the run time overhead.

8 Summary of the Method

Now we summarize briefly the main steps of query evaluation by dynamic filtering.

(i) *At system design time:*

- Create a system graph.

[3] Query graphs are undirected. SPGs are obtained from query graphs by replacing each edge with a pair of arcs pointing in opposite directions.

- Compute data sources for variables appearing in rel-nodes.

(ii) *At query compile time:*

- Initialize filters on the ports relevant to the query node.
- Construct SPG for each ax-node (SPG may depend on the query).

(iii) *At query run time:* All of the following actions may be performed in parallel and asynchronously, if desired.

- For each base relation select tuples allowed by the output filters, and send them through the appropriate ports.
- For each ax-node, whenever a set of new tuples arrive,
  (a) if possible, generate new tuples, and send them out to the succeeding rel-node;
  (b) propagate the data sideways (according to SPG) and update the corresponding filters;
- For each rel-node, whenever a file of new tuples arrives, transmit it further in accordance with the filters.
- For each filter, whenever a range of a variable x belonging to the filter changes, propagate it backwards to the filters occurring in $SRC(x)$.

(iv) *Termination:* Stop processing if no new tuple can be generated.

9 Comparison with Other Methods

Our current work was inspired by PROD and APEX [13, 12]. However, the filtering approach attacks the problem from a somewhat different angle and has much more potential than its ancestors. For instance, it can be extended to incorporate function symbols [10].

A few words about comparison between filtering and other methods are in order. Unfortunately, limitation of space does not allow us to present a detailed comparison. Several studies [1, 2, 11] suggest that apparently there does not exist a single method which in all cases is superior to the other methods. However, to give a rough idea about standing of the filtering method, we will briefly consider SNIP [14], Magic Sets [1], Counting [15] and the method of Henschen and Naqvi (abbr. H-N) [7].

The SNIP system of McKay and Shapiro has certain similar points with our method in that it is also based on data flow, and to that end, uses graph representation. Although the graph representation in SNIP is somewhat different from the system graph (see [11] for more details), this is by no means a main point. The major difference is that SNIP does not attempt at query optimization, and, therefore, cannot do better than the simplistic query evaluation (Section 5), which was the starting point of our optimization endeavour.

Magic Sets, Counting and H-N are compiled methods, and it is difficult to compare them with dynamic techniques in a general way. At the same time, there is a common feature: the filtering as well as Magic Sets and Counting determine sets of facts relevant to the query to optimize its evaluation. However, Magic Sets and Counting do this before the actual query evaluation starts, while dynamic filtering accomplishes this mainly at run time. It should be noted that, so far, filtering is more general than any of the aforesaid methods. For instance, H-N and Magic Sets are mainly restricted to the case of linear rules, Counting [15] runs into trouble when data stored in the database turns to be cyclic.

A more detailed comparison between (static) filtering and the above four methods is given in [11]. Our comparison based on the examples borrowed from [7, 11, 1] shows that in most cases filtering does as well as, and often better than, the three compiled methods. But Examples 4 and 6 from [1] are not in favour of the filtering in the version described in this paper. Nevertheless, in all such cases axioms of the "unfavourable" examples can be rewritten in a way that rehabilitates efficiency of the filtering. We conjecture that in many such interesting (but unfavourable) cases

the set of axioms can be (automatically) redesigned so that query evaluation under the filtering method will mimic (or improve upon) the strategy stemming from the compiled methods. Some results to that extent will be reported elsewhere.

As a matter of fact, although compiled methods have a different nature than that of dynamic ones, our stand is that these methods should not only be contrasted, but rather combined. Indeed, Magic Sets and Counting may produce an efficient set of rules employing the information available at query compile time. This set of rules can be then used as an input to a dynamic system such as the filtering, to take full advantage of the actually stored data at the query run time. Such a combined procedure would benefit from both the efficiency of compiled methods and the flexibility of dynamic ones.

10 Conclusions

The filtering method described in this paper is well competitive with the best known techniques for deductive query evaluation, even in their favourite domains. The main features of the filtering approach are:

(a) Universality. Filtering method, presented in this paper, can handle any function-free Horn databases. Incorporating of function symbols into this method is discussed elsewhere [10].

(b) The method takes advantage of actual data stored in the system.

(c) Like most bottom-up methods it is set-oriented (in contrast to the tuple-at-a-time processing, like in PROLOG).

(d) Preprocessing is used both at system design and at query compile time, reducing the run time overhead.

(e) The method naturally suggests parallel and distributed mode of processing by assigning a process to each node of the system graph.

(f) It combines top-down and bottom-up paradigms. Data sources and backward propagation are instances of the former, while data flow and fact generation represent the latter.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *ACM Symposium on Principles of Database Systems*, 1986.

[2] F. Bancilhon and R. Ramakrishnan. An amateur‘s introduction to recursive query processing strategies. In *ACM SIGMOD Conference on Management of Data*, pages 407–430, New York, 1986. ACM.

[3] J.S. Conery and D.F. Kibler. AND parallelism and nondeterminism in logic programs. *New Generation Computing*, 3:43–70, 1985.

[4] D. DeGroot. Restricted AND-parallelism. In *Proc. of the Int. Conf. on Fifth Generation Computer Systems*, pages 471–478, 1984.

[5] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

[6] G. Gardarin and C. DeMaindreville. Evaluation of database recursive logic programs as recurrent function. In *ACM SIGMOD Conference on Management of Data*, 1986.

[7] L.J. Henschen and S.A. Naqvi. Compiling queries in relational first-order databases. *Journal of ACM*, 31(1):47–85, 1984.

[8] Y.E. Ioannidis and E. Wong. An algebraic approach to recursive inference. In *Proc. of the 1st Expert Database Conf.*, pages 209–223, 1986.

[9] Y. Kambayashi. Processing cyclic queries. In W. Kim, D.S. Reiner, and D.S. Batory, editors, *Query Processing in Databases*, pages 62–80. Springer Verlag, Heidelberg, 1985.

[10] M. Kifer and E.L. Lozinskii. Can we implement logic as a database system? Technical Report TR 86/16, Dept. of Computer Science, SUNY at Stony Brook, June 1986.

[11] M. Kifer and E.L. Lozinskii. Filtering data flow in deductive databases. In *Int'l Conference on Database Theory*, volume 243 of *Lecture Notes in Computer Science*, pages 186–202, Rome, Italy, September 1986.

[12] E.L. Lozinskii. Evaluating queries in deductive databases by generating. In *Int'l Joint Conference on Artificial Intelligence*, pages 173–177, 1985.

[13] E.L. Lozinskii. A remark on distributed termination. In *ACM Transactions on Database Systems*, pages 323–356, September 1986.

[14] D.P. McKay and S.C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Int'l Joint Conference on Artificial Intelligence*, pages 368–374, 1981.

[15] D. Sacca and C. Zaniolo. The generalized counting method for recursive logic queries. In *Lecture Notes in Computer Science*, volume 243, pages 31–53, September 1986.

[16] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, MD, 1982.

[17] J.D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, pages 289–321, September 1985.

[18] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733–742, Oct. 1976.

[19] E. Wong and Youssefi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, September 1976.