Cost Evaluation of Methods for Query Processing in Deductive Database Systems

V. Iltchev

Key words: *Cost evaluation; cost metrics; deductive databases; algorithms.*

Abstract. *The paper describes: the important characteristics of an extensional database; a sample intensional database; some sample queries, whose processing will be evaluated; a cost metrics, based on the number of intermediate facts generated. On this basis, a cost comparison between well-known methods for query processing and a new bottom-up method, developed by the author, has been made.*

1. Introduction

The classical method of *asympthotic notation* for cost evaluation of algorithms **cannot** be applied for cost evaluation of methods for query processing in deductive database systems (DDBS), because the inference engine does **not** operate directly with the data structures, which hold the facts. Between the inference engine and the data structures lies an isolation layer. This layer is the relational database management system (RDBMS).

Therefore, the efficiency of a query processing method for DDBS can be evaluated only regarding the way this method formulates and orders the queries to the RDBMS. This affects the number of intermediate facts, the RDBMS must generate by answering these queries. The smaller this number is, the better the efficiency of the query processing method appears to be proves to be becomes. Recall, that not all intermediate facts take part in producing the final answer.

The paper describes:

– the important characteristics of an extensional database;

– a sample intensional database, used further for cost evaluation of the query processing methods;

– some sample queries, whose processing (translation) will be evaluated;

– the cost metrics, based on the number of intermediate facts generated.

On this basis, a cost comparison between well-known methods for query processing **and** a new bottom-up method, developed by the author, has been made.

2. Characteristics of the Extensional Database

The sample extensional database, used further, is shown on *figure 1*.

This directed graph can be stored as a binary relation, where domain elements hold information about nodes and tuples hold information about edges.

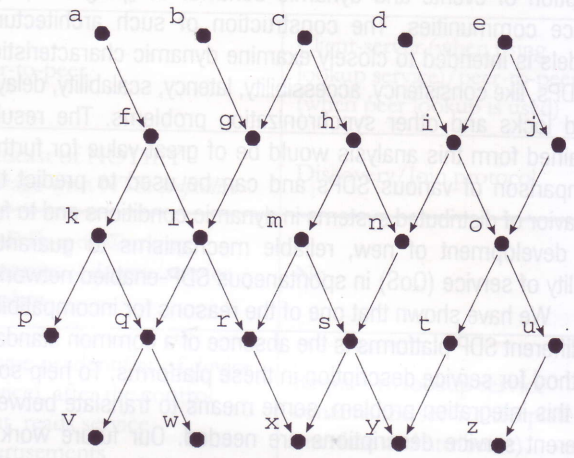

Figure 1

PAR(PARENT, CHILD) = {<a,f>, <b,g>, <c,g>, <c,h>, <d,i>, <e,i>, <e,j>, <f,k>, <f,l>, <g,l>, <h,m>, <h,n>, <i,n>, <i,o>, <j,o>, <k,p>, <k,q>, <l,q>, <l,r>, <m,r>, <m,s>, <n,s>, <o,t>, <o,u>, <q,v>, <q,v>, <r,x>, <s,x>, <s,y>, <t,y>, <u,z>} .

Apparently, there is a recursive closure between the attributes PARENT and CHILD. The reason to pick out such an example is to test how the differnt methods will process recursive sets of rules.

Additionally, nodes are arranged in layers. Each edge goes from a node of one layer into a node of the next layer. In other words, this formal model does not represent cycles and shortcuts. Nodes in the first layer have no incoming edges. Nodes in the last layer have no outgoing edges. All other nodes have at least one incoming and one outgoing edge.

A formal model of such type of data is shown on *figure* 2.

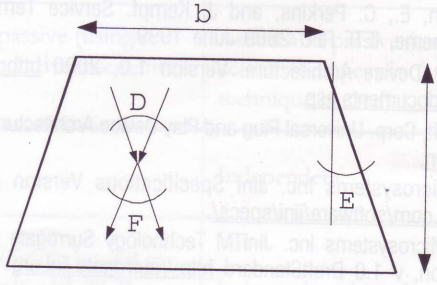

Figure 2

According to it, the important data charactics are:

– b – base. This is the number of nodes that have no antecedents.

– h – height. This is the length of the longest chain in R.

The number of layers is (h+1). They are numbered from *0* to *h*, where b is the number of nodes in layer *0*.

– D – duplication factor. This is the average number of incoming edges into a node.

– F – fan-out factor. This is the average number of outgoing edges from a node.

– E – expansion factor, calculated as E = F / D.

It is assumed that the data are random, with an uniform distribution. Thus, F and D obtain average values.

More detailed description of this model can be found in [1].

Additional definitions and derivations

Let gsum(E, h) denote the sum of the elements of a geometric series of ratio E, with length (h+1), thus:

$gsum(E, h) = 1 + E + E^2 + E^3 + ... + E^h$.

Let n(i) denote the number of nodes at level i. According to the definition for expansion factor:

$n(i+1) = n(i)*E$, and

$n(i) = b*E^i$.

Let N denote the total number of nodes. Then:

$N = b*(1 + E + E^2 + E^3 + ... + E^h) = b*gsum(E, h)$.

Apparently, the number of edges entering level i is n(i-1)*F, and the number of edges leaving level i is n(i)*F. Thus, the total number of edges, denoted as A, is:

$A = bF + bEF + bE^2F + bE^3F + ... + bE^{(h-1)}F$

$= bF(1 + E + E^2 + E^3 + ... + E^{(h-1)})$

$= b*F*gsum(E, h-1)$.

Let h' denote the average level. Then:

$$h' = h - \left\lfloor \left(\sum_{i=1}^{h} (i * n(i)) \right) / N \right\rfloor = h - \left\lfloor \left(\sum_{i=1}^{h} (i * b * E^i) \right) / N \right\rfloor .$$

Let *a* and *b* be nodes from different layers. If *b* is reachable from *a* then an *arc* exists between these two nodes. But *b* may be reachable from *a* through several *paths*. Even in such case there is only one *arc* between both nodes. In other words – fact *b* is derived from fact *a* only once.

The number of arcs of length k going from level i to level (i+k) is:

$n(i+k) = n(i)E^k$.

Let a(k) denote the number of all arcs of length exactly k in the whole binary relation. The value of a(k) is obtained by summing all the arcs of length k that enter level i for i = k to h. Thus:

$a(k) = n(k) + n(k+1) + ... + n(h)$

$= n(k)*(1 + E^1 + E^2 + ... + E^{h-k}) = n(k)*gsum(E, h-k)$

$= b*E^k*gsum(E, h-k)$.

Let P denote the total number of paths in the whole binary relation. In contrast to the number of arcs, here we can **not** use the number of nodes on an intermediate level, because each of these nodes has been reached via more than one paths, and on the current level each of these paths will be extended with F new edges. In other words – the duplication factor does not act in this case. Therefore, the total number of paths is:

$P = b*F*gsum(F, h-1)$.

3. Sample Intensional Database

Let par (X,Y) denote the extensional database predicate, which corresponds to the binary relation, shown on *figure 1*.

Set of rules I

anc(X,Y) :- par(X,Y).

anc(X,Y) :- par(X,Z), anc(Z,Y).

It defines the *ancestor* relationship as: X is an ancestor of Y **either** if X a parent of Y **or** if X is a parent of someone Z, who is an ancestor of Y.

The second rule has been rewritten without *left-recursion*, because the Prolog approach has also been choosen for the cost comparison. Prolog uses the back-tracking algorithm, which can make the inference engine fall into an infinite loop, while processing left-recursive rules.

Set of rules II

sgc(X,Y) :- eq(X,Y).

sgc(X,Y) :- par(X1,X), sgc(X1,Y1), par(Y1,Y).

It defines the *same-generation-cousins* relationship as: **either** X and Y is one and the same person **or** the parents of X and Y are same-generation cousins too.

The predicate eq(X,Y), can be implemented through a biuld-in function or through a relation in the database, with the following tuples:

EQ = {<a,a>, <b,b>, <c,c>, <d,d>, ..., <z,z>}.

4. Sample Queries

Query 1

:- anc(b,Y).

The adornment is anc^{bf}. Thus, the inference process follows the direction of data recursion.

Query 2

:- anc(X,v).

The adornment is anc^{fb}. Thus, the inference process goes in the opposite direction to the data recursion.

Query 3

:- sgc(v,Y).

The adornment is sgc^{bf}. Here the inference process goes up and down over the data graph.

5. Cost Metrics

As mentioned in the introduction, the inference engine does **not** operate directly with the data structures, which hold the facts. Between the inference engine and the data structures lies

an isolation layer, which is the RDBMS. Therefore, the efficiency of a query processing method for DDBS can be evaluated only regarding the way this method formulates and orders the queries to the RDBMS. The smaller the number of intermediate facts, the RDBMS must generate by answering the queries is, the better the efficiency of the query processing method.

Regarding this, the number of *successful inferences* (or *successful firings*) has been chosen as cost measure. For a rule in the form:

$p :- q_1, q_2, \ldots, q_n$

a successful firing is (id, t, t_1, t_2, ... , t_n), where t_1, t_2, ... , t_n are tuples in q_1, q_2, ... , q_n and t is a tuple in p. It denotes that the truth of t_1, t_2, ... , t_n is used to establish that t is true, by applying the given rule. The identifier id is used because it is possible, that the same inference is made repeatedly. Thus, the cost function measures the size of the intermediate results before duplicate elimination.

The conjunction of the predicates in the rule body is translated into a join-operation. By recursive rules, this join-operation is repeatedly applied to the temporal relation, corresponding to the recursive predicate. It is assumed, that the cost for each application is proportional to the size of the temporal relation. Hence, by measuring the size of this intermediate relation over all steps, a cost is obtained, that is proportional to the actual cost.

6. Corrections in the Formal Data Model

The formal data model and the corresponding method for cost evaluation, described in [1], give an incorrect answer by:

– stratified methods, applied to queries with bound arguments;

– the final selection which must be applied at the end of a non-stratified method.

For example, according to [1], the number of facts in the answer of query:

:- anc(b,Y).

is equal to the number of nodes in a tree, rooted at *b*, which is:

$E + E^2 + E^3 + \ldots + E^h = gsum(E, h)$.

The authors of [1] came to this result by simply taking the formula for the total number of nodes (see chapter 2) and substituting the *base* with 1. This is not correct because the formula for the total number of nodes is applicable for average values of F and D. But when we start from a single node, then, for the first several layers, we have $E \approx F$, because the duplication factor exerts influence on the formula early when the number of nodes, reached on the subsequent layer, increases to a significant value.

More frappant in [1] however is the case with the query:

:- anc(X,v).

where the authors assume that, because the inference process goes in the opposite direction, it is enough to substitute E with 1/E, in order to calculate the number nodes, rooted at *v*. Thus, this number shall be:

$$\frac{1}{E^2}+\frac{1}{E^3}+\frac{1}{E^4}+\ldots..+\frac{1}{E^h}.$$

But, this is a convergent series, which has as upper bound 1. Hence, at most one fact (one node) will be generated, which is fully incorrect.

Thefore in such cases (stratified methods or final selection), I propose to define two different expansion factors:

– E_{down} – when the inference process follows the expansion of facts;

– E_{up} – when the inference process goes in the opposite direction of the expansion of facts.

6.1. Defining E_{down}

The behaviour of E_{down} is illustrated on fig.3., i.e on the first step E_{down}=F, and then it decreases exponentially to F/D.

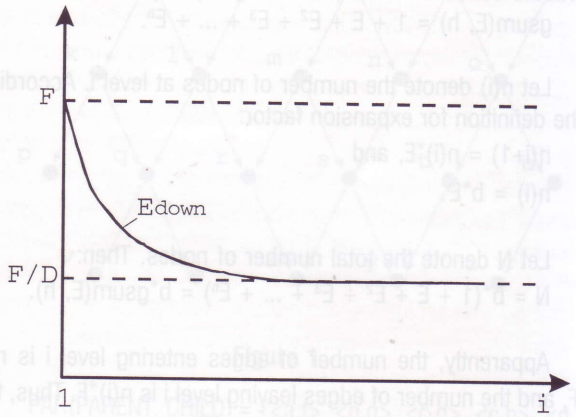

Figure 3. Behaviour of E_{down}

According to [10], an exponential function can be described as:

$f(x) = a^*e^{b^*(x-c)} + d$

where:

c – translates graph horizontally;

b – does horizontal stretching or compression;

a – does vertical stretching or compression;

d – translates graph vertically.

We can simplify this function by substituting c=0 and b=1, thus:

$E_{down}(i) = a^*e^i + d$.

To make it a decreasing function, we have to make the base e < 1, and because the decrease of E_{down} is determined by D, it is reasonable to substitute e = 1/D. The lower bound of E_{down} is F/D, therefore we have to substitute d = F/D. Thus:

$$E_{down}(i) = a^*\left(\frac{1}{D}\right)^i+\frac{F}{D}.$$

To determine a, we must consider that for $i=1 \rightarrow E_{down}=F$. Thus, after this substitution:

$$F=\frac{a}{D}+\frac{F}{D}.$$

Hence:

$$a = F^*(D-1).$$

Finally, after substitution of a, in the formula for E_{down}, we

receive:

$$E_{down}(i) = F*(D-1)*\left(\frac{1}{D}\right)^{i} + \frac{F}{D}.$$

Because we start from a single node (b=1), the number of nodes, reached on a particular level, is the product of the nodes, reached on the previous level, and E_{down}, thus:

$$n(i) = E_{down}(i-1) * E_{down}(i).$$

After substitution:

$$n(i) = \prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j} + \frac{F}{D}\right).$$

Thus, the total number of nodes reached is:

$$N = \sum_{i=1}^{h}\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j} + \frac{F}{D}\right).$$

which will be denoted from now on as esum(E_{down},h).

6.2. Defining E_{up}

The behaviour of E_{up} is illustrated on fig.4., i.e on the first step E_{up}=D, and then it decreases exponentially to 1.

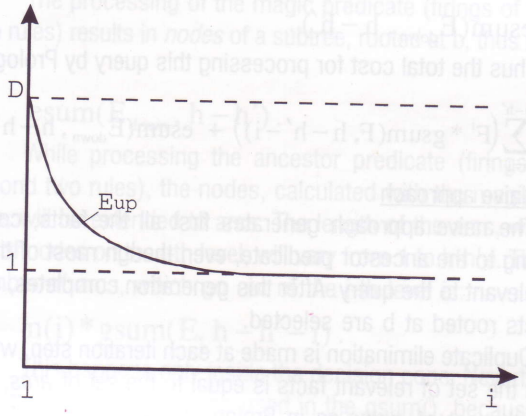

Figure 4. Behaviour of E_{up}

Although D/F < 1, the lower bound of E_{up} is 1. To explain the reason for this, the term *decision cone* has to be defined. According to [6], the decision cone has as apex a node, which has the value of a bound query argument. The decision cone comprises all the nodes (facts) from different layers, rooted at this *apex node*. Because D/F < 1 it is possible, that some nodes inside the decision cone will not be reached, but this will not shrink the decision cone, because the neighbour nodes will turn into apexes, hence *D* instead of *E* will act on them. Thus, on its next layer, the decision cone will be dense again.

To describe E_{up} we will use the same simplified exponent function as in chapter 6.1.:

$E_{up}(i) = a*e^{i} + d.$

To make it a decreasing function, we have to make the base e < 1, and since the decrease of E_{up} is determined by F, it is reasonable to substitute e = 1/F. The lower bound of E_{up} is 1, therefore we have to substitute d = 1. Thus:

$$E_{up}(i) = a*\left(\frac{1}{F}\right)^{i} + 1.$$

To determine a, we must consider that for i=1→E_{up}=D. Thus, after this substitution:

$$D = \frac{a}{F} + 1.$$

Hence:

$$a = F*(D-1).$$

Finally, after the substitution of a in the formula for E_{up}, we receive:

$$E_{up}(i) = F*(D-1)*\left(\frac{1}{F}\right)^{i} + 1.$$

Since we start from a single node (b=1), the number of nodes, reached on a particular level, is the product of the nodes, reached on the previous level, and E_{up}, thus:

$$n(i) = E_{up}(i-1) * E_{up}(i).$$

After substitution:

$$n(i) = \prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j} + 1\right).$$

Thus, the total number of nodes reached is:

$$N = \sum_{i=1}^{h}\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j} + 1\right).$$

which will be denoted from now on as esum(E_{up},h).

7. Methods, Chosen for Competitive Cost Evaluation

7.1. Prolog

Prolog has been chosen as a typical top-down method. It performs classical back-tracking by processing recursive sets of rules. This makes the information retrieval *tuple-at-a-time* oriented, which is very inefficient for large databases. Moreover, rules with left recursion make the inference engine fall into an infinite loop.

The back-tracking approach in Prolog is described in [8]. An implementation, using relational algebra, can be found in [5]. Another implementation of this method is the Java Jedd-engine, described in [12].

7.2. Naive Approach

This is a bottom-up strategy where the rule set is translated into SQL-expressions [15]. In case of recursive rule set, an iterative algorithm, which implements the *least fixed-point* strategy [14] and [16], has to be applied. Such an algorithm will generate, as first, all possible facts (most of them irrelevant to the query), and then a final selection will be applied, to produce the answer. That is very inefficient for large databases. Addi-

tional characteristic for the naive approach is, that the whole quantity of facts, generated at all previous steps, is used to generate facts for the current iteration step. The naive approach is described in [2].

7.3. Semi-naive Approach

The only difference of this approach, in comparison with the naive approach is, that only facts generated on the last-previous iteration step are used to generate the facts for the current iteration step. This saves time and memory. A proof of correctness of the semi-naive approach is given in [6].

7.4. Magic Sets Transformation Method

It belongs to the group of „rewriting methods", called so, because they exploit the goal structure, to transform an inefficient program into a „smart" one, written in the same logic programming language. The idea behind the Magic Sets Transformation Method is: to find first all possible values that can ever appear on the places of bound arguments. To achieve this, the rule set has to be extended with additional predicates, called also „magic predicates". These predicates generate the values for the bound arguments. A complete description of the „Magic Sets Transformation Method", as well of other methods from the same group, like „Counting Method", „Reverse Counting Method" etc. is given in [3] and [4].

The rewriting methods are the most effective and widespreading methods. Some implementations of their are described in [13,7,9] and [17].

In most cases the „Counting Method" and the „Reverse Counting Method" are more effective than the „Magic Sets Transformation Method". But these two methods are **not** general, which means, that there are private cases of rule sets, which can make the inference engine fall into an infinite loop. Because of this, the „Magic Sets Transformation Method" has been chosen for a comparison, as a common method, applicable for all linear sets of rules.

7.5. 2P-Method

This is a new bottom-up method, developed by the author of this paper and published in [11]. The deduction process goes through two phases: *expand phase* and *shrink phase* (thus, the name comes from – 2P-method). At each iteration step of these two phases a differential (set of new facts and eventually new requests) is computed and stored in the predicate's temporal relation for further use.

An expand phase differential is the result of *Sideways Information Passing (SIP)* of the requests, generated at the previous iteration step. This differential is a union of two sets:

– set of facts generated through propagating to base conjunctions;

– set of requests generated through *SIP*, where the distinguished arguments come from a base conjunction build only from the distinguished database predicates in the rule body. The undistinguished arguments are substituted with the **NULL** value.

The shrink phase differential is computed by replacing the occurrences of recursive predicates in rule-bodies with the facts already generated during the expand phase. Thus, the whole rule-body becomes a base conjunction, which generates new facts. Expand phase differentials are used subsequently and in reverse order for substitutions during the shrink phase.

8. Cost Evaluation

8.1. Query:

:- anc(b,Y).

Prolog

Because of the back-tracking, Prolog does not make duplicate elimination, which means that instead of arcs, all the paths of different length rooted on b will be generated. Recall, that the number of all paths of length k is:

p(k) = b*F^k*gsum(F, h-k).

Since all these paths are rooted on one single node, the base b is 1.

Usually, a query has as a bound arguments facts, which do not lie on the base layer, but on an intermediate one. Therefore h' has to be used instead of h. Thus, the sum of all paths with length from h' to h is:

$$\sum_{i=1}^{h-h'} \left(F^i * \mathrm{gsum}(F, h-h'-i) \right).$$

To eliminate the duplicate facts in this result, a final selection must be made. It will generate the number of nodes rooted at b, which is:

$$\mathrm{esum}(E_{down}, h-h').$$

Thus the total cost for processing this query by Prolog is:

$$\sum_{i=1}^{h-h'} \left(F^i * \mathrm{gsum}(F, h-h'-i) \right) + \mathrm{esum}(E_{down}, h-h').$$

Naive approach

The naive approach generates first all the facts, corresponding to the ancestor predicate, even though most of them are irrelevant to the query. After this generation completes, only the facts rooted at b are selected.

Duplicate elimination is made at each iteration step, which means the set of relevant facts is equal to the set of *arcs*, and not to the set of *paths* as in Prolog. Important for the naive approach is, that the whole set of facts (generated on all previous iteration steps) takes part by computing the facts at the current step. As a consequence, at the second step, when the arcs of length 2 are computed, the arcs of length 1 will be recomputed again, thus the total cost for this step is (a(1) + a(2)). At the third step, arcs of length 1 and 2 will be recomputed, thus the cost is (a(1) + a(2) +a(3)), and so on. Hence, the total number of arcs generated is:

$$D * \sum_{i=1}^{h} \left((h-i+1) * a(i) \right).$$

The arcs generation starts from the base layer, and not from an intermediate one. Therefore h has been used instead of h'.

Recall, that the number of all arcs of length k is: a(k) = b*E^k*gsum(E, h-k). The cost for the final selection is the same like in Prolog. Thus, the total cost by the naive approach for processing this query is:

$$D*\sum_{i=1}^{h}\left((h-i+1)*b*E^{i}*gsum(E,h-i)\right)+esum(E_{down},h-h').$$

Semi-Naive approach

The only difference between the naive and the semi-naive approach is, that by the semi-naive approach only the facts, generated at the previous iteration step, take part by computing the facts of the current step. Thus, the total number of arcs generated is:

$$D*\sum_{i=1}^{h}a(i).$$

The total cost for the semi-naive approach, including the final selection, is:

$$D*\sum_{i=1}^{h}\left(b*E^{i}*gsum(E,h-i)\right)+esum(E_{down},h-h').$$

Magic Sets Method

The rewritten system for this query is:

magic(b).
magic(Y) :– magic(X), par(X,Y).
anc(X,Y) :– magic(X), par(X,Y).
anc(X,Y) :– magic(X), par(X,Z), anc(Z,Y).

The processing of the magic predicate (firings of the first two rules) results in *nodes* of a subtree, rooted at b, thus the cost is:

$$esum(E_{down},h-h').$$

While processing the ancestor predicate (firings of the second two rules), the nodes, calculated with the magic predicate, will be extended to arcs. The length of the arcs, which go out of nodes on the i-th level, will vary from 1 to h-h'-i. Thus, the number of arcs, which go out of the i-th level is:

$$n(i)*gsum(E,h-h'-i).$$

This happens only inside the decision cone. Nevertheless, E instead of E_{down} has been used in the gsum(), because most of the arcs are with small length, and they are placed near the cone base, which means that E instead of E_{down} will act on them. As mentioned in 6.1, the number of nodes on the i-th level is:

$$n(i)=\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right).$$

Hence, the whole number of arcs inside the decision cone is:

$$\sum_{i=1}^{h-h'}\left(\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right)*gsum(E,h-h'-i)\right).$$

The total cost for the magic sets method, including the final selection, is:

$$\sum_{i=1}^{h-h'}\left(\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right)*gsum(E,h-h'-i)\right)+2*esum(E_{down},h-h')$$

2P-Method

During the expand phase all the nodes in the decision cone will be generated. The number of requests generated gets near the number of nodes. Therefore the cost for the expand phase is a sum of these two numbers, thus:

$$2*esum(E_{down},h-h').$$

During the shrink phase, nodes, calculated during the expand phase, will be extended to arcs. As by the magic sets method this happens only inside the decision cone, but with the difference of an additional selection at each iteration step, made by using the requests in the subsequent expand phase differential. The effect of this selection is: at each step the arcs to the current level will be extended to the nodes of the next level, and not newly generated as by the magic sets method. Hence, we will compute nodes rather than arcs. Therefore, the cost for the shrink phase is only:

esum(E_{down}, h - h').

Since a final selection is not necessary, the total cost is:

3*esum(E_{down}, h - h').

After substitution of esum(), the total cost is:

$$3*\sum_{i=1}^{h-h'}\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j}+1\right).$$

Conclusions for query:

:- anc(b,Y).

It is obvious that if a query starts from an intermediate level, then the stratified methods will show better performance than the unstratified ones. Therefore, in order to create uniform conditions for all methods, we will assume that each query starts from the first (or resp. the last) layer and spreads over the whole depth of database. Thus, the cost formulas are:

Prolog:

$$\sum_{i=1}^{h}\left(F^{i}*gsum(F,h-i)\right)+esum(E_{down},h).$$

Naive approach:

$$D*\sum_{i=1}^{h}\left((h-i+1)*b*E^{i}*gsum(E,h-i)\right)+esum(E_{down},h).$$

Semi-naive approach:

$$D*\sum_{i=1}^{h}\left(b*E^{i}*gsum(E,h-i)\right)+esum(E_{down},h).$$

Magic Sets Method:

$$\sum_{i=1}^{h}\left(\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right)*gsum(E,h-i)\right)+2*esum(E_{down},h).$$

2P-Method:

$$3*\sum_{i=1}^{h}\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j}+1\right).$$

Thus, we can conclude that:

1. For large base and small depth of recursion, Prolog has better performance than the naive and the semi-naive approaches,

because $F^i < b^*E^i$.

2. For real databases, where F << b:

$$\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right) < b^*E^i.$$

Hence, the magic sets method has a better performance than naive and semi-naive approaches.

3. The magic sets method has always a better performance than Prolog, because:

$$\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right) < F^i.$$

Recall that each member of the left series is equal to or smaller than F (see definition of E_{down} in 6.1).

The right member of the sum for total costs of Prolog, and of naive and semi-naive approaches is:

esum(E_{down},h)

where for the magic sets method it is:

2*esum(E_{down},h)

But, this will not affect vastly the total sum, because its left member has the form:

$$\sum_{i=1}^{h}\left(\prod_{j=1}^{i}(...)*\sum_{j=1}^{h-i}(...)\right).$$

By analogy with the terminology of *asymptotic notation* we should say that the *order of growth* of this formula is n^2.

On the contrary, the right member of the total sum, the esum(), has the form:

$$\sum_{i=1}^{h}\left(\prod_{j=1}^{i}(...)\right)$$

which has an *order of growth* - n.

Please consider that by this reasoning we exclude the member:

$$\prod_{j=1}^{i}(...)$$

because it presents in both formulas.

4. The 2P-Method has a better performance than the other four methods because, if we apply the terminology of *asympthotic notation* again: the 2P-Method has an *order of growth* - n, while all other methods have *order of growth* – n^2. The physical explanation is, that the 2P-Method computes *nodes*, while all other methods compute *arcs* or *paths*.

8.2. Query:

:- anc(X,v).

Prolog

Prolog will generate each path in the data structure on fig.1 and then check to see if it ends on 'v'. Recall, that the number of all paths of length k is: $p(k) = b^*F^k*gsum(F, h-k)$. Hence, the total number of paths, which may enter node 'v' is:

$$\sum_{i=1}^{h}\left(b*F^i*gsum(F,h-i)\right).$$

In the final selection E_{up} has to be used instead of E_{down}:

esum(E_{up},h')

because its direction is opposite to the transitive closure of facts. Thus, the total cost is:

$$\sum_{i=1}^{h}\left(b*F^i*gsum(F,h-i)\right)+esum(E_{up},h').$$

Naive approach

The naive approach does not consider the query bindings. Therefore, as by the previous query, the method will generate the whole amount of arcs and then select only those, which answer the query. The only difference is, that the final selection uses E_{up} instead of E_{down}, for the same reason as in Prolog. Thus, the total cost is:

$$D*\sum_{i=1}^{h}\left((h-i+1)*b*E^i*gsum(E,h-i)\right)+esum(E_{up},h').$$

Semi-Naive approach

Only facts, generated at the previous iteration step, take part by computing the facts of the current step. Thus, the total cost for the semi-naive approach, including the final selection, is:

$$D*\sum_{i=1}^{h}\left(b*E^i*gsum(E,h-i)\right)+esum(E_{up},h').$$

Magic Sets Method

The rewritten system for this query is:

magic(v).

anc(X,Y) :– magic(Y), par(X,Y).

anc(X,Y) :– magic(Y), par(X,Z), anc(Z,Y).

The magic predicate generates only one fact. It is v. This fact will be substituted in the two rules for the ancestor predicate. Because the anwer set for the magic predicate contains only one fact, at each iteration step the arcs to the current level will be extended to the nodes of the next level, and not newly generated. Hence, we will compute *nodes* rather than *arcs*. Thus, the cost is only:

esum(E_{up},h').

In this case a final selection is not necessary, but the magic sets algorithm can not distinguish it. Therefore it applies in principle a final selection to each rule set. Hence, the total cost, including this final selection, is:

1+2*esum(E_{up},h').

2P-Method

The expand phase lasts only one single step. Hence, only facts, connected to the query through *edges*, are generated. The cost for them is:

D

At each step of the shrink phase these edges will be extended to the nodes of the next level. Hence, we will compute *nodes* rather than *arcs*. Thus, the cost is only:

esum(E_{up},h').

A final selection is not necessary, thus the total cost is: $D + esum(E_{up},h')$.

Conclusions for query:

:- anc(X,v).

As by the previous query, in order to create uniform conditions for all methods (stratified and unstratified), we will assume that each query starts from the first (or resp. the last) layer and spreads over the whole depth of database. Thus, the cost formulas are:

Prolog:

$$\sum_{i=1}^{h}\left(b*F^i*gsum(F,h-i)\right)+esum(E_{up},h).$$

Naive approach:

$$D*\sum_{i=1}^{h}\left((h-i+1)*b*E^i*gsum(E,h-i)\right)+esum(E_{up},h).$$

Semi-naive approach:

$$D*\sum_{i=1}^{h}\left(b*E^i*gsum(E,h-i)\right)+esum(E_{up},h).$$

Magic Sets Method:

$1+2^*esum(E_{up},h')$.

2P-Method:

$D+esum(E_{up},h')$.

Thus, we can conclude that:

1. For big values of both F and D, Prolog becomes less effective than the naive approach, because F >> E.

2. The semi-naive approach has always a better performance than Prolog, because $b^*E^i < b^*F^i$. The physical explanation is, that the semi-naive approach does a duplicate elimination at each step. Thus it computes *arcs*, while Prolog computes *paths*.

3. The magic sets method and the 2P-Method have better performance than Prolog, naive approach and semi-naive approach, because if we apply the terminology of *asympthotic notation* again: magic sets method and 2P-Method have *order of growth* – n, while the other three methods have *order of growth* – n^2. The physical explanation is, that magic sets method and 2P-Method compute *nodes*, while the other three methods compute *arcs* or *paths*.

4. The 2P-Method is a little more effective than the magic sets method:

$D + esum(E_{up},h') < 1 + 2^*esum(E_{up},h')$

because it does no final selection.

8.3. Query:

:- sgc(v,Y).

The authors of [1] assume that the join $par_{up}.sgc.par_{down}$ is generated as follows:

1. Obtain a tuple from the current relation for sgc.
2. Look for matching tuples in par_{up}.
3. Look for matching tuples in par_{down}.
4. Take the cartesian product of the first column of the tuples from par_{up} and the second column of the tuples in par_{down}.

For this purpose they define $T_{A,B}$ as transfer ratio between two sets A and B. It is the fraction of these nodes in A which also appear in B, hence $0 \le T \le 1$. For example, if R and S are relations, where i is the i-th attribute of R and j is the j-th attribute of S, then $n*T_{R_i,S_j}$ gives the number of tuples in the result of the join operation between R and S on R_i and S_j.

We will simplify this model of evaluation assuming that the relational expression $par_{up}.sgc.par_{down}$ is evaluated as a whole, because:

1. This is an internal operation of the RDBMS and we can not control it.

2. This evaluation procedure will be applied to all methods analyzed, i.e. all the methods will be evaluated under same conditions.

Prolog

Prolog will generate all paths of different length in up-direction, which start from the query node, and extend each of these paths with paths in down-direction, whose length will

$$\sum_{i=1}^{h'}\left(D^i*gsum(D,h'-i)*\sum_{j=1}^{i}\left(F^j*gsum(F,i-j)\right)\right).$$

vary from 1 to the length of the path in up-direction. The deduction process will start from an intermediate level with base b = 1, therefore the costs are:

To eliminate duplicate facts in this result, a final selection must be made, but its cost is insignificant. Moreover, it presents as cost in each method, except the 2P-Method. Therefore it will be not included in cost formulas.

Naive approach

Each fact is build by two arcs, which start from same node and have same length.

As mentioned in 6.1, if we start from a single node, then the number of nodes reached on the k-th level is:

$$n(k)=\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right).$$

Each pair of these nodes builds a fact, i.e. the number of facts on this k-th level is a *Cartesian product* of the set of nodes with itself, thus:

$$\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2.$$

For a node on i-th level of the data structure, the maximal length of an arc, which can generate facts, is k = h - i. Hence, the number of all facts, generated from a node on the i-th level, is a sum up to this length, thus:

$$\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2.$$

The number of facts, generated from all the nodes on i-th level of the data structure is:

$$b*E^i*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2.$$

Because of recomputing, the number of facts for all levels is:

$$\sum_{i=1}^{h}\left[(h-i+1)*b*E^i*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2\right].$$

Semi-Naive approach

The semi-naive approach does not recompute arcs, therefore the total cost is:

$$\sum_{i=1}^{h}\left[b*E^i*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2\right].$$

Magic Sets Method

The rewritten system for this query is:

magic(v).
magic(X) :– magic(Y), par(X, Y).
sgc(X,Y) :– magic(X), eq(X,Y).
sgc(X,Y) :– magic(X), par(X1, X), sgc(X1, Y1), par(Y1, Y).

The processing of the magic predicate (firings of the first two rules) results in *nodes* of an inverted subtree (a subtree in up-direction), rooted at the query node. Thus the cost is:

esum(E_{up}, h').

While processing the same-generation-cousins predicate (firings of the second two rules), each node, calculated with the magic predicate, will be extended with arcs. Each pair of these arcs, which have the same length and start from one same node, will build an answer.

Following the pattern for naive and semi-naive approaches, the number of facts, generated from one single node on i-th level of the magic set, is:

$$\sum_{k=1}^{h'-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2.$$

Hence, the number of facts, generated from all the nodes on i-th level of the magic set, is:

$$\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^j+1\right)*\sum_{k=1}^{h'-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2.$$

Thus, the total number of facts is:

$$\sum_{i=1}^{h'}\left[\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^j+1\right)*\sum_{k=1}^{h'-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2\right]+\text{esum}(E_{up}, h').$$

2P-Method

During the expand phase all the *nodes* of an inverted subtree, rooted at the query node, will be generated. The number of requests generated, nears the number of nodes. Therefore the cost for the expand phase is:

2*esum(E_{up}, h').

Recall that during the shrink phase a nested *SELECT* operation is used to separate only those facts, which correspond to the requests generated during the subsequent step of the expand phase. The purpose is, at each shrink phase step there have to be selected only these nodes, which lie on a fixed distance from the apex of the decision cone, the distance is determined by the step number. Therefore, the cost for the shrink phase is only:

$$\prod_{j=1}^{h'}\left(F*(D-1)*\left(\frac{1}{F}\right)^j+1\right)*\sum_{k=1}^{h'}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right).$$

In other words, we first find the nodes on the base layer, reachable from the query node in up-direction (see the left multiplier) and then extend these nodes with arcs in down-direction to the layer with the query node (see the right multiplier). Additionally, the selection upon the requests in the subsequent expand phase differential removes the cost for the Cartesian product from the formula.

Thus, the total cost for the 2P-Method is:

$$\prod_{j=1}^{h'}\left(F*(D-1)*\left(\frac{1}{F}\right)^j+1\right)*\sum_{k=1}^{h'}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)+2*\text{esum}(E_{up}, h')$$

Cost for final selection

It is:

$$E_{up}^{h'}*E_{down}^{h'}$$

Which may be written also as:

$$\prod_{i=1}^{h'}\left(F*(D-1)*\left(\frac{1}{F}\right)^i+1\right)*\prod_{i=1}^{h'}\left(F*(D-1)*\left(\frac{1}{D}\right)^i+\frac{F}{D}\right).$$

In other words, we first find the nodes on the base layer, reachable from the query node in up-direction and then use them to find the nodes on the layer with the query node. Because of the selection with the query node, no Cartesian product is necessary.

Conclusions for query:

:- sgc(v,Y).

As by the previous queries, in order to create uniform conditions for all methods (stratified and unstratified), we will assume that each query starts from the first (or resp. the last) layer and spreads over the whole depth of database. Thus, the cost formulas are:

Prolog:

$$\sum_{i=1}^{h}\left(D^i*\text{gsum}(D, h-i)*\sum_{j=1}^{i}\left(F^j*\text{gsum}(F, i-j)\right)\right).$$

Naive approach:

$$\sum_{i=1}^{h}\left[(h-i+1)*b*E^i*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2\right].$$

Semi-naive approach:

$$\sum_{i=1}^{h}\left[b*E^i*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^j+\frac{F}{D}\right)\right)^2\right].$$

Magic Sets Method:

$$\sum_{i=1}^{h}\left[\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j}+1\right)*\sum_{k=1}^{h-i}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right)\right)^{2}\right]+\text{esum}(E_{up},h)$$

2P-Method:

$$\prod_{j=1}^{h}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j}+1\right)*\sum_{k=1}^{h}\left(\prod_{j=1}^{k}\left(F*(D-1)*\left(\frac{1}{D}\right)^{j}+\frac{F}{D}\right)\right)+2*\text{esum}(E_{up},h)$$

Thus, we can conclude that:

1. For small base and big values of both F and D, Prolog becomes less effective than the naive approach.

2. The semi-naive approach has always a better performance than Prolog, because the member of the nested sum by the semi-naive approach is smaller than those by Prolog, and because the first two members of the outher sum by the semi-naive approach are smaller than these by Prolog.

3. For real databases, where F << b:

$$\prod_{j=1}^{i}\left(F*(D-1)*\left(\frac{1}{F}\right)^{j}+1\right) < b*E^{i}.$$

Hence, the magic sets method has a better performance than the semi-naive approach.

4. The 2P-Method has a better performance than the other four methods because, if we apply the terminology of *asymptothic notation* again: the 2P-Method has an *order of growth* - n, while all other methods have *order of growth* - n^2. The physical explanation is, that the 2P-Method computes *nodes*, while all other methods compute *arcs* or *paths*.

9. Graphical Cost Representation

The graphics are generated with MathLab by interpreting the cost formulas, derived above. The generation has been made by a constant base and an expansion factor, and by a variable height.

Query 1

:- anc(b,Y).

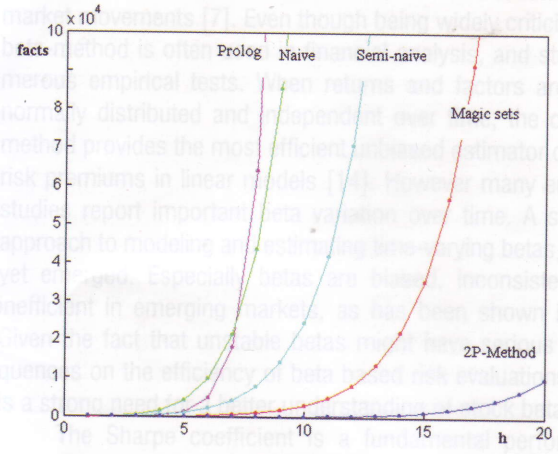

Figure 5

Query 2

:- anc(X,v).

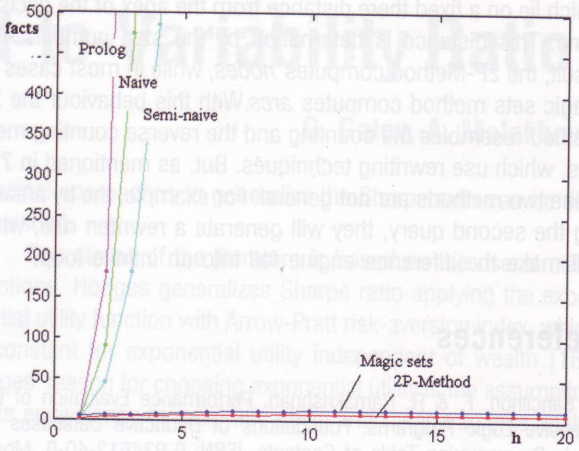

Figure 6

Query 3

:- sgc(v,Y).

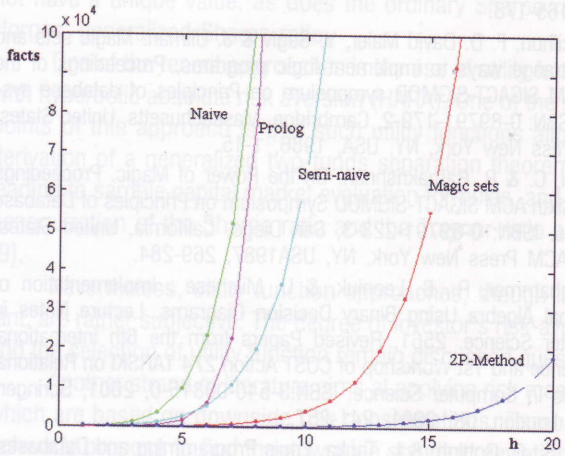

Figure 7

10. Conclusions and Future Work

1. For a small base and big values of both F and D, the naive and the semi-naive approaches show a better performance than Prolog, even though it is a stratified method. This is because the naive and the semi-naive approaches do a duplicate elimination on each step. Thus, they compute *arcs*, while Prolog computes *paths*.

2. The magic sets method has a better performance than Prolog, because as a bottom up method it does a duplicate elimination on each step. Thus, it computes *arcs*, while Prolog computes *paths*. The magic sets method has a better performance than the naive approach and the semi-naive approaches, because it uses stratification, based on a rewriting technique, which allows to find at first all possible values that can ever appear on the places of bound arguments.

3. The 2P-Method has a better performance than the magic sets method because of the additional selection, implemented

by computation of the shrink phase differentials. The purpose is, at each shrink phase step have to be selected only these nodes, which lie on a fixed there distance from the apex of the decision cone. This distance is determined by the step number. As a result, the 2P-Method computes *nodes*, while in most cases the magic sets method computes *arcs*.With this behaviour the 2P-Method resembles the counting and the reverse counting methods, which use rewriting techniques. But, as mentioned in 7.4., these two methods are **not** general. For example, the by answering the second query, they will generate a rewriten rule, which will make the inference engine fall into an infinite loop.

References

1. Bancilhon, F. & R. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. Foundations of Deductive Databases And Logic Programming Table of Contents. ISBN: 0-934613-40-0, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988, 439 - 517.

2. Bancilhon, F. Naive Evaluation of Recursively Defined Relations, Springer Topics In Information Systems, on Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies. ISBN:0-387-96382-0, Springer-Verlag New York Inc., New York, NY, USA, 1986, 165-178.

3. Bancilhon, F. D. David Maier, Y. Sagiv & J. Ullman. Magic sets and other strange ways to implement logic programs, Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems. ISBN: 0-89791-179-2, Cambridge, Massachusetts, United States, ACM Press New York, NY, USA, 1986, 1-15.

4. Beeri, C. & R. Ramakrishnan. On the Power of Magic. Proceedings of the Sixth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. ISBN: 0-89791-223-3, San Diego, California, United States, 1987, ACM Press New York, NY, USA1987, 269-284.

5. Berghammer, R., B. Leoniuk, & U. Milanese . Implementation of Relational Algebra Using Binary Decision Diagrams. Lecture Notes in Computer Science. 2561, Revised Papers from the 6th International Conference and 1st Workshop of COST Action 274 TARSKI on Relational Methods in Computer Science. ISBN:3-540-00315-0, 2001, Springer-Verlag, London, UK, 2001, 241-257.

6. Ceri, S.; G. Gottlob. & L. Tanka. Logic Programming and Databases. Springer-Verlag, ISBN 3-540-51728-6, Berlin, 1990.

7. Chien-Le G.241-257, A. Kazuki, T. Masahiko & N. Shojiro. Database Compression with Data Mining Methods. The Kluwer International Series in Engineering and Computer Science, Logic-based Artificial Intelligence. ISBN:0792372247, 2000, Kluwer Academic Publishers, Norwell, MA, USA, 2000, 177-190.

8. Clocksin, W. & C. Melish. Programming in Prolog. Using the ISO Standard. Springer-Verlag, ISBN 3-540-00678-8, Berlin, 2003.

9. Greco, S. & E. Zumpano. On the Rewriting and Efficient Computation of Bound Disjunctive Datalog Queries, Proceedings of The 5th Acm Sigplan International Conference on Principles and Practice o Declaritive Programming. ISBN: 1-58113-705-2, Uppsala, Sweden, 2003, ACM Press, New York, NY, USA, 2003, 136-147.

10. Husch, L.,S. Transformations of Exponential Functions. Lecture Notes 2001. http://archives.math.utk.edu/visual.calculus/0/shifting.5/index.html.

11. Iltchev, V. Bottom-up Method for Processing Recursive Sets of Rules. International Conference on Computer Systems and Technologies. ISBN 954-9641-33-3, ACM-acmbul&UAI, Sofia, Bulgaria, 19-20 June, 2003, II.18.1-II.18.6.

12. Lhotak, O. & L. Hendren. Jedd: a BDD-based Relational Extension of Java. Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. ISBN:1-58113-807-5, Washington DC, USA, 2004, ACM Press, New York, NY, USA, 2004, 158-169.

13. Mumick, S.,I. & H. Pirahesh. Implementation of Magic-sets in a Relational Database System. Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. ISSN: 0163-5808, Minneapolis, Minnesota, United States, 1994, ACM Press, New York, NY, USA, 1994, 103-114.

14. Tarski, A. Lattice-theoretic Fixpoint Theorem and its Applications.– *Pacific journal of Mathematics*, 1955, No 5, 285-309.

15. Ullman, J. Implementation of Logical Guery Languages for Databases– *ACM Transactions on Database Systems (TODS)*. 10, September, 1995, No 3, 289-321, ISSN: 0362-5915.

16. Van Emden, M. H. & R. A. Kovalski . The Semantics of Predicate Logic as a Programming Language.– *Journal of the ACM (JACM)*, 23, October 1976, No 4, 733-742, ISSN: 0004-5411, ACM Press New York, NY, USA.

17. Wenfei, F., X. Y Jeffrey, L Hongjun, L. Jianhua & R. Rajeev . Query Translation from XPATH to SQL in the Presence of Recursive DTDs. Proceedings of the 31-th International Conference on Very Large Data Bases. ISBN 1-59593-154-6, Trondheim, Norway, VLDB Endowment, August 30 - September 02, 2005, 337-348.

Manuscript received on 14.09.2006

Velko Iltchev *received the MSc degree in Radioelectronics from the Technical University in Sofia, Bulgaria, in 1987. Since 1987 he is Assistent Professor in the Department of Computer Systems and Technologies at the Technical University – Sofia, branch Plovdiv. He specialized in the Universities of Regensburg, Passau and Karlsruhe, with grants from the EU and from the German Academic Exchange Service. He is currently in procedure of receiving a PhD degree. His research interests are in: language processors (compiler construction), relational and deductive databases, XML strorage and retrieval in/from relational databases, in which areas he has over 20 publications.*

Contacts:
Department of Computer Systems
Technical University – Plovdiv
e-mail: iltchev@tu-plovdiv.bg