1-1-2008

Implementation of recursive queries for information systems

Jayalakshmi Jeyaraman
University of Nevada, Las Vegas

IMPLEMENTATION OF RECURSIVE QUERIES FOR INFORMATION SYSTEMS

by

Jayalakshmi Jeyaraman

Bachelor of Engineering in Computer Science
Sri Venkateswara College of Engineering, India
June 2006

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2008

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

JULY 7TH , 2008

The Thesis prepared by

JAYALAKSHMI JEYARAMAN

Entitled

IMPLEMENTATION OF RECURSIVE QUERIES FOR INFORMATION SYSTEMS

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Implementation of Recursive Queries for Information Systems

by

Jayalakshmi Jeyaraman

Dr. Kazem Taghva, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Sophisticated information systems require a powerful query language and an efficient implementation strategy. In practice, these information systems are either built on the top of an existing database management system or built as an expert system with deductive capabilities. Both of these implementations must provide a mechanism to express recursive queries. It is therefore a necessity for the system to have an efficient algorithm to evaluate these queries. In this thesis, we give a detailed description of a bibliographic database, a set of recursive queries, an overview of some standard query processing algorithms, and an implementation of these queries in DATALOG.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my committee chairman and advisor Dr. Kazem Taghva, through whose patience, understanding, and valuable advice, this work has been accomplished. I would also like to express my gratitude to Dr.Ajoy K Datta who helped me in all my endeavors at UNLV and Dr. Wolfgang Bein ,with whom I have had many different interesting conversations when I served as his TA for being my committee members. I would like to thank Dr Emma Regentova who has accepted my request to serve as my graduate college representative

I would like to thank my husband & my loving family for their encouragement and motivation. I would like to thank my friends for their support. To all of them, I dedicate this work.

CHAPTER 1

INTRODUCTION

Deductive database systems are those which express queries by means of logic rules. These database systems may be viewed as an advanced form of relational database systems. At present most of the information systems are built on top of these systems as they are more expressive and provide better features that support recursive queries. Relational database systems are not that expressive and do not have mechanisms that support recursive query processing. Evaluating queries, in particular the recursive queries of deductive database systems is an open challenge. Datalog is the language typically used to specify facts, rules and queries in deductive databases. Deductive databases try to combine logic programming with relational databases. Deductive databases are more expressive than relational databases but less expressive than logic programming systems. But the advantage of using Datalog over logic programming is that it does not process one tuple at a time as logic programming does, rather it processes a set of tuples at a time. To evaluate recursive queries using Datalog we need to know the basics of logic databases. Most of the examples in this chapter are from the article "An Amateur's Introduction to Recursive Query Processing Strategies" by Francois Bancilhon and Raghu Ramakrishnan

1.1 Logic databases

Deductive or logic databases have become the main field of research in recent times. The main features of these systems are (i) capability to express queries by means of logical rules (ii) provide efficient algorithms to evaluate recursive queries (iii)provide efficient optimization techniques. A database is a set of unordered rules. Given a database we can partition it into a set of rules and a set of facts. The set of facts are known as extensional database and a set of rules are known as intensional database. Deductive databases also divide their information into two categories namely, data and rules. Data or facts are represented by a predicate with constant arguments. For example the fact 'parent (cain, adam)' means that Adam is the parent of Cain. Here 'parent' is the name of a predicate, and the fact 'parent (cain, adam)' is represented extensionally, that is, this is a true value that is stored in the database. Rules are generally represented as

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

Here p and the q_i's are literals. A literal is of the form p (t_1, $t_2 \ldots\ldots t_n$) where 'p' is a predicate of arity 'n' and each t_i is a constant or a variable .Here 'p' is called the head of the rule, and each of the qi's is called a goal. The conjunction of the qi's is the body of the rule .For example "uncle (john, X) - brother(X, Y), parent (john, r) "is a rule with head "uncle (john, X)" and body "brother(X, Y), parent (john, Y)". A ground clause is a rule in which the body is empty and a fact is a ground clause with no variables. A predicate whose relation is stored in the database is called as Extensional Database (EDB).

1.2 Syntax of a logic database

There are four types of names associated with logic database. They are the (i) variable names (ii) constant names (iii) predicate or relation names and (iv) evaluable predicate names. The syntax for naming these variable names is as follows. Variable names are a string of characters starting with upper case letter and the other characters are either upper or lower case letters. Constants are a string of characters starting with lower case letters or integers. For example X and Y are variables whereas abel and adam are constants. Predicate names and relation names are denoted by identifiers starting with lower case letters. The term relation is from database terminology and it is interpreted by a set of tuples and predicate is from logic terminology and it is interpreted by a true/false function. There is a fixed arity associated with each relation/predicate. An instantiated literal is one that does not contain any variables. For example

"id (john, 25) "is an instantiated literal whereas "id (john, age) "is not.

If p (t_1, t_2.....t_n) is a literal, we call (t_1, t_2.....t_n) a tuple.

1.3 Interpretation of a logic database

We have till now seen the syntactical explanation of logic databases. Now we move on to the semantic interpretation. We try to associate a set of instantiated tuples with each relation name. We assume that with each evaluable predicate 'p' is associated a set natural (p) of instantiated tuples which we call its natural interpretation. For example, an infinite set of 3-tuple (x, y, z) of integers can be associated with predicate 'sum' such that the sum m of x and y is z can

have infinite values.

A model of a database is obtained by assigning truth values to all variables that makes all rules true. For example consider the following set of rules

p(x):- q(x)

q(x):- r(x)

Assume that r (1), q (1), p (1), p (2), q (2), p (3) are true and all others are false, is a valid model. The assumption r (1) is true and all others are false is an invalid model. The interpretation of this is that, in a model if the right hand side is true then the left hand side is also true. So assume values to variables that make all rules true. A rule can be understood as if the body of the rule is true then the head is also true. For a given database, there may be many models, but a nice property of Horn clause is that there is only one minimal model which we call as the model of that database. A minimal model is a model such that none of its subset is a model. Therefore a model or an interpretation of a database always means the minimal model of the database. In this example r (1), q (1), p (1) is the minimal model for r (1).

Next we shall see what adornment of a predicate is. Let p be an n-ary predicate. An adornment of p is a sequence 'a' of length 'n' of b's and f's. For example bbf is an adornment of a ternary predicate. An adornment is to be interpreted as follows, ith variable of 'p' is bound (respectively free) if the ith element of 'a' is b (respectively f). We denote adornments by superscripts. A query form is represented as id^{bfb}.

1.4 An example

To understand more let us look into a logic database. The facts and the rules of a logic database are given as:

Facts:

parent (cain, adam)

parent (abel, adam)

parent (cain, eve)

parent (abel, eve)

Rules:

ancestor (X, Y) – ancestor(X, Z), ancestor (Z, Y)

ancestor (X, Y) – parent(X, Y)

generation (adam, 1)

generation (X, I) - generation(Y, J), parent(X, Y), J=I-1

generation (X, I) – generation (Y, J), parent(Y, X), J= I+1

In this database, parent, ancestor and generation are the set of predicates or relation names. J=I+1 and J=I-1 are arithmetic predicates, cain, adam, eve and abel are constants. X, Y, Z are variables and "parent (cain, adam)" is a fact, and "ancestor(X, Y) - parent(X, Y)" is a rule. We now try to associate meaning with the database. We try to map the constants to a real world objects. Imagine Abel to be the name of a person. The arithmetic predicates are mapped to their respective arithmetic operators. We can intuitively interpret each fact and each rule. For Instance we interpret the fact "parent (cain, adam)", by saying that the rule parent hold for cain and adam and we interpret the rule "ancestor (X, Y) -

ancestor(X, Z), ancestor (Z, Y)" by saying that if there are three objects X, Y and Z such that if "ancestor(X, Z)" is true and "ancestor (Z, Y)" is true then "ancestor(X, Z)" is true. Then we associate with each predicate a set of tuples. Now we have to answer queries of the form ancestor (abel, X). For this we have to know the structure and representation of the logic database and understand what recursion is. They are explained in the next sections.

1.5 Structuring and representing the database

There are several ways of representing the logic database. A predicate that appears only in the intensional database is referred as derived predicate. A predicate that appears only in the extensional database or in the body of the rule is knows as base predicate. Any given database can be modified into its equivalent containing only base and derived predicates. Having done this there are different methods of representing the set of rules, here we choose rule/goal graphs. This graph has two set of nodes square nodes that are associated with predicates, and oval nodes that are associated with the rules. If rule is of the form

$$r:\ p - p_1, p_2 \ldots\ldots\ldots\ldots p_n$$

in the intensional database, then there is an arc going from node r to node p, and for each predicate p1 there is an arc from node p1 to node r.

For example consider the rules

r1:p1 – p3, p4

r2:p2 – p4, p5

r3:p3 – p6, p4, p3

r4:p4 – p5, p3

r5:p3 – p6

r6:p5 – p5, p7

The rule/goal graph is given in Figure 1.1. Here we can clearly see that there is an arc from r1 (p1-p3, p4) represented in an oval to p1 represented in a square. This is known as a rule/goal graph. Now we have to know what recursive queries are in order to solve them. In the next section we will see what recursion is and how the logic databases that involve recursive queries are represented using rule/goal graph.

1.6 Dependency graph and recursion

It is necessary to understand how the predicates in a logic program depend on one another. Dependency graph exhibits the dependency among the predicates. The nodes of the graph correspond to the predicates and there is an arc from predicate p to predicate q if there is a rule whose head is predicate p and body is predicate q. Hence the presence of a loop in the dependency graph suggests that the rule is recursive. For non recursive rules the graph is acyclic. Recursive rules are those that involve recursion. We say a rule is recursive if it is of the form,

ancestor (X, Y) - ancestor (X, Z), parent (Z, Y)

We say that a rule is linear, if it is recursive and the recursive predicate appears only once on the right hand side. This is sometimes referred to as regularity. For example sg (X, Y) – p(X, XP), p(Y, YP), sg(XP,YP)

is linear and

ancestor (X, Y) – ancestor (X, Z), ancestor (Z, Y)

is non-linear. Now consider another database with rules

p (X, Y) – a1(X, Z),q (Z, Y)

q (X, Y) – p(X, Z), a2 (Z, Y)

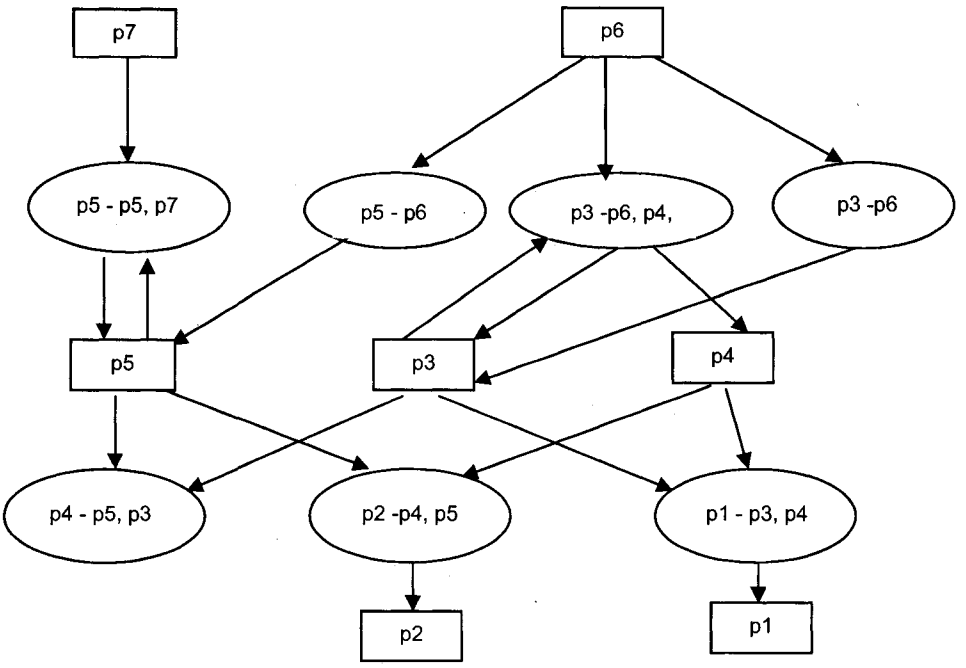

Figure 1. 1 Rule/goal Graph

According to the above discussions they are not recursive but we can clearly see that both predicates p and q are recursive. So in a multi-rule context if p and q are the two predicates, we say p derives q if p→q occurs in the body of the

rule whose head predicate is q. A predicate p is said to be recursive if p→+p. Two predicates p and q are mutually recursive if p→+q and q→+p. Thus we say that two predicates may be mutually recursive if and only if the predicates in their heads are mutually recursive. So now we can modify the rule/goal graph to describe the non-recursive part by grouping the mutually recursive predicates and isolating the recursive parts. Now the squares will be associated with non-recursive predicates or with blocks of mutually recursive predicates and oval nodes are associated with non-recursive rules or with blocks of mutually recursive rules. The representation for the previous database is given in Figure 1.2.

1.7 Properties of queries

Safety and range restriction are the two properties of queries. Given a database and a set of queries we always want to ensure that the queries are safe. It is undesirable to have unsafe queries. If q is a set of queries in a database D, we say that q is safe in D if the answers to q are finite. There are two kinds of unsafe queries

(i)The arithmetic predicates are often unsafe. Consider the query "greater than (27, X)", is unsafe as X can have infinite number of values as answer.

(i)The rules with free variables in the head which do not appear in the body are unsafe. For example a query "likes (joe, X)" is unsafe because, in the minimal model of the database "likes (joe, X)" is true for every integer X.

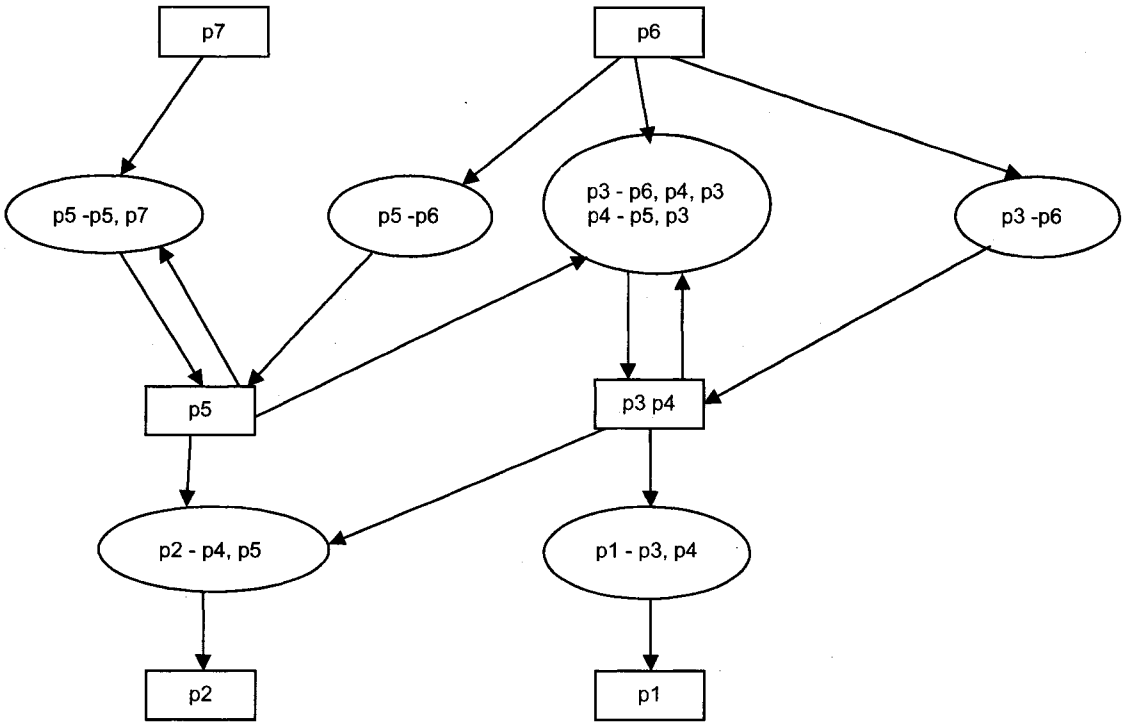

Figure 1. 2 Simplified Rule/goal Graph

Next we will see what is range restricted. A rule is range restricted if every variable of the head appears somewhere in the body. For example "likes (joe, X)" is not range restricted. A set of rules will be range restricted if every rule in this set is range restricted. If all the evaluable predicates have finite set of tuples associated with it then it is guaranteed to be safe.

Hence before we try to evaluate the queries we have to ensure that they are safe and finite. Thus we have seen the syntax and the semantics of logic databases, their interpretation, an example to explain them in detail, the structure and representation of these databases, what is recursion and safety of queries.

Given the logic database the problem now is to answer recursive queries of the form “ancestor (cain, X)”. For this we have to know the different evaluation and optimization techniques .The following chapter will give a detailed explanation of the available techniques and how each one of them is better than the other.

CHAPTER 2

EVALUATION AND OPTIMIZATION OF RECURSIVE LOGIC QUERIES

Safe queries are not guaranteed to be evaluable. We can in fact specify that the user should ensure that his query is safe. There are different strategies to deal with logic queries. Strategies are defined based on the application domain and the algorithms that are available to reply to queries. The first class of strategies consists of actual evaluation algorithms. Given a query it gives an answer to the query, for example naive evaluation and semi-naive evaluation. The second class assumes that the underlying evaluation is either naive or semi-naive and then optimizes the rules to make their evaluation more efficient for example, counting and reverse counting and magic sets. We know how to evaluate SQL queries. Converting an SQL query to a datalog program will help us to understand datalog programming better. In this chapter let us see how to convert an SQL query to a datalog program, and then we will look into fixed point evaluation of datalog queries and then naive, semi-naive and magic set evaluation. Most of the examples in this chapter are from the book "Principles of Database and Knowledge-base systems" by J.D.Ullman.

2.1 Converting an SQL query to datalog program

Consider two relations

(i)Beers (name, manufacturer)

(ii)Sells (bar ,beer ,price)

The first relation contains the name and manufacturer of beer. The second relation details about the bars that sell beer and their corresponding prices. Query is to find the manufacturers of the beers that Joe sells. In SQL it is expressed as

```
SELECT manufacturer
FROM Beers
WHERE name IN (
SELECT beer
FROM Sells
WHERE bar = 'Joe's Bar'
);
```

In datalog this is expressed as

JoeSells (b) →Sells ('Joe's Bar', b, p)

Answer (m) → JoeSells (b), Beers (b, m)

Here Beers and Sells are Extensional Database (EDB), Answer and JoeSells are Intensional Database (IDB).

2.2 Fixed points of datalog equations

Fixed points of datalog equations are obtained when we substitute values for the predicates such that the body and the head of the equation are equal.The fixed points are not always unique for a given equation. We have already seen in

section 1.3 about model and the minimal model of a database. So the unique minimal model that contains the EDB's is the unique minimal fixed point with respect to those EDB's.

Let there exist relations R1, R2....Rk with EDB predicates r1, r2, r3....rk and set of IDB predicates p1, p2, p3....pm with variables P1, P2, P3 ...Pm. Now to obtain the fixed point assign the EDB relations R1, R2.....Rk to the IDB variables P1, P2, P3.....Pm such that the equations are satisfied. Suppose there are two solutions to the equations then we should be able to form some form of logical relationship between them. If S1 and S2 are the two solutions and if S2<= S1 then S2 is a subset of S1. In general we can say S0 as the minimal fixed point if there is no other S, such that S<=S0. If there is no such S0, then the equation does not contain least fixed point.

Let us consider an example to understand this. A graph is represented by an EDB predicate arc(X, Y); arc(X, Y) is true if and only if there is an arc from X to Y. We have the following set of rules

path (X, Y):- arc (X, Y)

path (X, Y):- path(X,Z), path (Z,Y)

Here the first rule states that a path can be a single arc and the second rule states that the concatenation of two paths yields to a path. We can convert these rules into a single equation

$P(X, Y) = A(X, Y) \cup \pi_{X,Y}(P(X, Z) \bowtie P(Z, Y))$

Now if the nodes are {1, 2, 3} and there is an arc from 1 to 2 and from 2 to 3 then A= {(1, 2), (2, 3)} .From rule 1 we can say that P = {(1, 2), (2, 3)} and from

the rule 2, (1, 3) is added to P. So now P = {(1, 2), (2, 3), (1, 3)} is a solution to the above equation. Let us consider the right hand side of the equation,

$\pi_{X, Y}(P(X, Z)) \bowtie P(Z, Y))$

When we substitute the values, {(1, 2), (2, 3), (1, 3)}$\bowtie$ {(1, 2), (2, 3), (1, 3)}, the join on the right gives a tuple (1, 2, 3) over the distribution list (X, Z, Y) and the projection of X, Y on this yields (1, 3), and the union over A gives {(1, 2), (2, 3), (1, 3)} which is equal to L.H.S. Thus {(1, 2), (2, 3), (1, 3)} = {(1, 2), (2, 3)} U $\pi_{X, Y}$ ({(1, 2), (2, 3), (1, 3)} {(1, 2), (2, 3), (1, 3)}).On the other hand when P = {(1, 2), (2, 3), (1, 3), (3,1)}, the join on the right yields a tuple (3,2) that is not there in the left hand side so this is not a solution.

Next we will look into a model that is not a fixed point. Consider P= {(1, 2)}. Here the two rules are true irrespective of the values substituted for them. But P (1, 2) does not satisfy the equation. Therefore P= {(1, 2)} is not a fixed point of the equation with respect to EDB A=null.

2.3 Top down vs. bottom up

The evaluation strategies are classified into top-down and bottom-up. The top-down or backward chaining strategy starts with the query as a goal and expands from the head to the body of the rule and forms more goals. The beauty is that none of these goals formed are irrelevant to the query. However some of the goals may lead us to a point where we cannot proceed further. This happens because the possible solutions to the query may not be there in the database. The bottom-up or forward chaining strategy starts from bodies of the rules to their

heads and continue evaluating until the required query is generated. Top-down may be efficient as the query is known but they are very complex. Bottom-up on the other hand are simpler, but they evaluate a lot of useless results as they do not know what they are evaluating. The bottom-up evaluation ensures that the set of values for body variables is finite at each step; however there may be infinite number of steps. For recursive queries bottom-up evaluation proves to be better since each step produces a finite answer and we can make use of already computed values. Both these evaluation techniques in fact do the following

(i)generate the goals

(ii)while the goals are generated, evaluate them against the rules and

(iii)At each step, check for the termination conditions

Termination condition is reached when the new goal generated is empty or it has been already evaluated. For recursive queries bottom-up approach serves to be better. So we try to evaluate the recursive queries using bottom-up approach (naive and seminaive) and optimize using magic sets. Now let us see how the naive, seminaive and magic set works.

2.4 Naive evaluation

Naive evaluation is a bottom-up approach. For a given set of rules and a query, start with a rule where the predicate of the query is the head of the rule and the body of the rule is a base predicate i.e. an EDB relation whose value is stored in the database. Let us see an example to understand better

Facts:

parent (a, aa)

parent (a, ab)

parent (aa, aaa)

parent (aa, aab)

parent (aaa, aaaa)

Rules and query:

r1:ancestor(X, Y) – parent(X, Z), ancestor (Z, Y)

r2:ancestor(X, Y) – parent (X, Y)

r3:query (X) – ancestor (aa, X)

The datalog equation for these set of rules is given as

$A(X, Y) = P(X, Y) \cup \pi_{X,Y}\{A(X, Y) \bowtie A(Z, Y)\}$

Here we need to evaluate the rules to find the ancestor of aa. The algorithm is

begin

initialize ancestor to the empty set,

evaluate (ancestor (X, Y) –parent (X, Y)),

insert the result in ancestor,

while "new tuples are generated" do

begin

evaluate (ancestor (X, Y)—parent(X, Z), ancestor (Z, Y))

using the current value of ancestor, insert the result in ancestor

end

evaluate (query (X) – ancestor (aa, X)) and insert the result in query

end

Solving using the above algorithm for the given facts and the rules:

Step 1: Apply r1

ancestor = { (a,aa), (a,ab), (aa, aaa) ,(aa, aab),(aaa, aaaa) }

query = {}

Step 2: Apply r2

Evaluating the whole set of ancestors the following tuples are generated.

ancestor = {(a, aaa), (a, aab), (aa, aaaa)}

And the resulting state is

ancestor = {(a,aa), (a, ab), (aa, aaa),(aa, aab), (aaa, aaaa) ,(a, aaa),

(a, aab), (aa, aaaa)}

query = {}

Step 3: Apply r2

Again evaluating the whole set of ancestor the following tuples are generated {(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

The new state is

ancestor = { (a,aa),(a,ab),(aa, aaa), (aa, aab) ,(aaa,aaaa) ,(a, aaa),(a, aab) ,(aa, aaaa), (a, aaaa)}

query= {}

Because (a, aaaa) is new, we continue

Step 4: Apply r2

Again evaluating the whole ancestor set the following tuples are generated

{(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

Because there are no more new tuples the state does not change and we move to r3.

Step 5: Apply r3

Next we evaluate the query rule and the following tuples are produced in query {(aa, aaa), (aa, aaaa)}. Now the ancestor becomes

ancestor= {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (a, aaa), (a, aab), (aa, aaaa) ,(a, aaaa)}

query = {(aa, aaa), (aa, aaaa), (aa, aab)}

the algorithm terminates.

2.5 Semi-naive evaluation

Semi-naive is very similar to the naive evaluation except that it tries to reduce the number of duplications. At looping it tries to be smarter. The basic mechanism is that, it tries to evaluate only new tuples that are generated rather than evaluating the whole set of tuples. This tries to remove the disadvantage of bottom-up approach of generating useless tuples.

Now we shall look into the optimization techniques. The main drawback of naive evaluation method is

(i)Relevant facts are too big

(ii) A lot of duplicate computations are generated

For example consider the facts and rules used in the above section. Let us evaluate it using semi-naive method. The main principle of this method is the evaluation of the differential of the obtained set rather than the whole set.

The datalog equation of the above example is

A(X, Y) = P (X, Y) U $\pi_{X, Y}$ {A (X, Y) join A (Z, Y))}. Semi naive is nothing but incremental evaluation of least fixed points of this equation. Let us see how we perform semi-naive evaluation for the above example.

Step 1: Apply r1

ancestor = { (a, aa), (a, ab), (aa, aaa) ,(aa, aab),(aaa, aaaa)}

query = {}

Step 2: Apply r2

Evaluating the ancestor set we have:

d_ancestor_1 = {(a, aaa), (a, aab), (aa, aaaa)}

old_ancestor_1 is

old_ancestor_1 = {(a,aa), (a,ab), (aa, aaa) ,(aa, aab), (aaa,aaaa)}

new_ancestor_1=old_ ancestor_1 U dancestor_1

new_ancestor_1 ={(a, aa), (a, ab), (aa, aaa),(aa, aab), (aaa, aaaa),(a, aaa), (a, aab), (aa, aaaa)}

query = {}

Step 3: Apply r2

Here we no more evaluate among the old_ancestor_1 set. We evaluate new_ancestor_1 and the dancestor_1, and the tuple generated is

dancestor_2= {(a, aaaa)}

new_ancestor_2 =new_ancestor_1 U dancestor_2

new_ancestor_2 ={(a, aa),(a, ab),(aa, aaa), (aa, aab) ,(aaa, aaaa) ,(a, aaa),(a, aab) ,(aa, aaaa), (a, aaaa)}

query= {}

Step 4: Apply r2

Now we evaluate new_ancestor_2 and dancestor_2 and we get d_ancestor_3 .It is as an empty set.

Step 5: Apply r3

Next we evaluate the query rule and the following tuples are produced in query {(aa, aaa), (aa, aaaa)}. Now ancestor becomes

ancestor= {(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (a, aaa), (a, aab), (aa, aaaa), (a, aaaa)}

query = {(aa, aaa), (aa, aaaa), (aa, aab)}

algorithm terminates.

2.6 Comparison between naive and semi-naïve

The above relation can be expressed

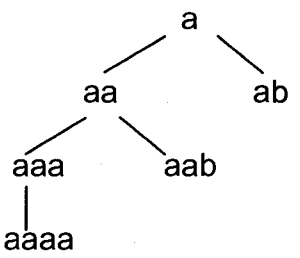

Figure 2. 1 Parent/Ancestor Relationships

Facts Rules and query

parent (a, aa)r1: ancestor(X, Y) – parent(X, Z), ancestor (Z, Y)

parent (a, ab)r2: ancestor(X, Y) – parent (X, Y)

parent (aa, aaa)r3: query (X) – ancestor (aa, X)

parent (aa, aab)

parent (aaa, aaaa)

Let us see how naive and semi naive evaluation work on these.

NaiveEvaluation Seminaive Evaluation

Step (1) Step (1)

ancestor = { (a,aa), (a,ab), ancestor = { (a,aa), (a,ab),

(aa, aaa) ,(aa, aab),(aaa,aaaa) } (aa, aaa) ,(aa, aab),(aaa,aaaa) }

Step (2) Step (2)

Iteration1: Iteration1:

ancestor = {(a, aaa),(a, aab) , ancestor = {(a, aaa),(a, aab) ,

(aa, aaaa)} (aa, aaaa)}

Iteration 2: Iteration 2:

{(a, aaa), (a, aab), (aa, aaaa),{(a, aaaa)}

(a, aaaa)}

Iteration 3: Iteration 3:

No new tuples No new tuples

2.7 Magic sets

The main idea of magic sets is to define a filter table that computes all

relevant values and restrict the computation to infer only tuples with relevant values in the first column. When the queries contain bound arguments, magic set is the best optimization technique. It tries to restrict the bottom-up evaluation of a logic program to those facts that are "potentially relevant" with respect to the query. A magic set transformation starts with a datalog program, a query with bound arguments, and an order to pass the query binding recursively from the rule head to body. Within a rule body, sideway information passing will occur for a fixed ordering of subgoals. A magic predicate is then defined for each of the differently bound version of a subgoal predicate so that only the tuples those are possible for the bound arguments are computed. Let us first see what is side way information passing (SIP).

2.7.1 Sideway information passing (SIP)

It is the decision on how to pass information sideways in the body of the rule when we are evaluating the rule. It specifies how the bindings in the head of the rule will be used and the order in which the sub goals in the body will be evaluated and how bindings will be passed between the predicates in the body.

2.7.2 Magic sets transformations

The idea behind the magic sets technique is to compute an auxiliary predicate called "magic_predicate" for each intensional database predicate in the original program. The magic predicate collects the bindings from all runtime goals for the associated predicate. The rules of the program are rewritten using the magic predicates so that "irrelevant" tuples are not generated during a bottom-up evaluation. A tuple is considered irrelevant if it is not an answer to any runtime

goal. Thus magic-sets transformation makes a bottom-up evaluation as efficient as top-down by avoiding the generation of irrelevant tuples. The magic sets transformation is defined on adorned programs (explained in section 1.3) and is guided by SIPs. For a given adorned program, an adorned query goal q^{α}, and full SIPs for each rule the magic sets produces a magic program as follows

(i)Create a new magic predicate "magic_predicate" for each derived predicate in adorned program.

(ii) For each rule r in adorned program, add a modified version of r to magic program. If rule r has head p (t), where t represents all arguments for the head predicate p, then the modified version is obtained by adding magic_predicate (t^b) into the body of r, where t^b denotes all bound arguments of p (t).

(iii) For each rule r in adorned program with head p (t) and for each subgoal q_i (t_i) where q is a derived predicate, add a magic rule to magic program. The head is mq_i (t_i^b). The body contains the literal magic_predicate(t^b) and all the subgoals preceded by q in the SIPs order associated with r.

(iv) Create a fact mq (c), where c is the set of constants equated to the set of bounded arguments.

Let us see how to perform magic transformation for the above example using the above mentioned steps.

The set of rules are

r1: ancestor(X, Y) – parent(X, Z), ancestor (Z, Y)

r2: ancestor(X, Y) – parent (X, Y)

r3: query (X) – ancestor (aa, X)

The adorned program and the adorned query are

r1: $ancestor^{bf}$ (X, Y) – parent(X, Z), $ancestor^{bf}$(Z, Y)

r2: $ancestor^{bf}$ (X, Y) – parent (X, Y)

adorned query is q^{bf} . The magic program is

r1: $ancestor^{bf}$ (X, Y) – $magic\_ancestor^{bf}$ (X), parent(X, Z), $ancestor^{bf}$(Z, Y)

r2: $ancestor^{bf}$ (X, Y) – $magic\_ancestor^{bf}$ (X), parent (X, Y)

r3: $magic\_ancestor^{bf}$ (Z) - $magic\_ancestor^{bf}$ (X), parent (X, Y)

r4: $magic\_ancestor^{bf}$ (aa)

Now we try to find software that will help us to perform the evaluation and optimization of the recursive queries for a given database. There are many developments in the field of deductive database. The paper "A survey of research on deductive database "by Raghu Ramakrishnan and Jeffrey D Ullman suggests many projects that implement these techniques. Some of them are CORAL, ADITI, XSB, and ConceptBase. Each of these was developed by different people at different universities. Out of these ConceptBase is the one that uses SLDNF strategy with a cache system that works similar to the bottom-up approach for evaluation of recursive queries and magic sets for optimization. The next chapter explains in detail about ConceptBase.

CHAPTER 3

CONCEPT BASE

ConceptBase started its development in 1987 at the Universities of Passau and Aachen. Versions have been distributed for research experiments since early 1988. The stable distribution versions are V3.3, V4.1, V5.2 and V6.1 that have been installed in more than five hundred sites worldwide and are seriously used by a dozen research projects in Europe and the America. Conceptbase seeks to combine deductive rules with semantic data model based on Telos. We saw what deductive are rules in our previous chapters. Semantic data models are those models that describe the database in terms of the kinds of entities that exist in the database, their grouping and structural interconnections among them. We will see what Telos is in this chapter. Conceptbase also provides support for integrity constraint. It has been used in a number of applications at various universities in Europe, and now being developed commercially. This chapter contains most of the details from the site http://dbis.rwth-aachen.de/CBdoc/.

3.1 What is ConceptBase?

ConceptBase is a deductive object base management system based on Telos data model. Telos is a conceptual modeling language that makes it well-

suited for design and modeling applications. The key features distinguishing ConceptBase from other extended DBMS and expert systems shells are:

- clean formal integration of deductive and object-oriented abstraction
- client-server architecture with wide-area Internet access
- equivalent logical, semantic network, and text frame representations

ConceptBase implements a version of the knowledge representation language Telos, which combines properties of deductive and object-oriented languages. Let us see Telos in detail in next section.

3.2 The telos language

Telos is a formal language for representing knowledge in a wide area of applications. It integrates object-oriented and deductive features into a logical framework. It is an experimental deductive object base management system, based on Telos data model. Telos is structurally object-oriented framework generalizes earlier data models and knowledge representation formalisms, such as entity-relationship diagrams or semantic networks, and integrates them with predicative assertions and temporal information. This combination of features seems to be particularly useful in software information applications such as requirements modeling and software process control. The following example is used throughout this section to illustrate the language:

Company has employees, some of them being managers. Employees have a name and a salary which may change from time to time. They are assigned to

derived from his department and the manager of that department.Thus the recursive queries like finding the boss of an employee can be easily done.

3.3 Frame and network representation

Telos supports three different representation formats: logical, graphical (semantic network) and a frame representation. Graphical and semantic formats are based on the logical one. Logical representation also forms the base for integrating a predicative assertion language for deductive rules, queries, and integrity constraints into the frame representation.

Telos knowledge base (KB) is a finite set of interrelated propositions or objects.

$$KB = \{P (oid, x, l, y, tt) \mid oid, x, y, tt \in ID, l \in LABEL\}$$

where oid has key property within the knowledge base, ID is a non-empty set of identifiers with a non-empty subset LABEL of names. The components oid, x, l, y, tt are called identifier, source, label (or name), destination and belief time of the proposition.

The object x has a relationship called l to the object y. This relationship is believed by the system for the time tt. As shown below there is a natural interpretation of a set of propositions as a directed graph (semantic network). They distinguish four patterns of propositions and give them the following names:

(i)Individuals

P (oid, oid, l, oid, tt)

("oid is an object with name l believed tt")

(ii) InstanceOf P (oid,*instanceof,y,tt)

("x is an instance of class y believed tt")

(ii)IsA

relationships (specializations)

P(oid, x,*isa, y, tt)and

(``x is a specialization of y believed tt")

(iii)Attributes

(All other propositions)

As a user, you don't work directly with propositions but with textual (frame) and graphical (semantic networks) views on them. Both are not based on the oid's of objects but on their label components. To guarantee a unique mapping we need the following naming axiom.

3.3.1 Naming axiom

The label ("name") of an individual object must be unique. The label of an attribute must be unique within all attributes with a common source object.

In this section we introduce it by modeling the employee example:

Individual Employee in Class with

attribute

name: String;

salary: Integer;

dept: Department;

boss: Manager

end

Individual Manager in Class isA Employee end

Individual Department in Class with

attribute

head: Manager

end

Individual Mary in Manager, Token with

name

hername: "Mary Smith"

salary

earns: 15000

dept

advises: PR;

currentdept: RD

end

Individual PR in Department, Token end

Individual RD in Department, Token end

The next frames establish two departments labelled PR and RD and state that the individual object “Mary” is an instance of the class Manager. Mary has four of attributes labelled hername, earns, advises and currentdept which are instances respective attribute classes of Employee with labels name, salary and dept.

3.3.2 Specialization axiom

The destination (“superclass") of a specialization inherits all instances of its

source ("subclass"). All instances of Manager including "Mary" are also instances of Employee. Telos enforces the attribute values by the following general axiom:

3.3.3 Instantiation axiom

If 'p' is a proposition that is an instance of a proposition 'P' then the source of 'p' must be an instance of the source of 'P', and the destination of 'p' must be an instance of the destination of 'P'.

For example, "Mary Smith" must be an instance of String. The individual "Mary" also shows another feature: attribute classes specified at the class level do not need to be instantiated at the instance level. This is the case for the boss attribute of Employee. On the other hand, they may be instantiated more than once as e.g. dept

Telos treats all three kinds of relationships (attribute, isa, in) as objects. Thus each attribute, instantiation or generalization link of Employee may have its own attributes and instances. For example, each of the four Employee attributes is an instance of an attribute class denoted by the label attribute but can also have instances of its own. The attribute with label earns of "Mary" is an instance of attribute salary of class Employee. Syntactically, attribute objects are denoted by appending the attribute label with an exclamation mark to the name of some individual. The relationship between salary and earns could be expressed as

Attribute mary!earns in Employee!salary

end

Instantiation links are denoted by "->" and specialization links by "=>":

InstanceOf mary->Manager

```
end
IsA Manager=>Employee
end
```

The operators can be combined to complex expressions. The following example shows how to reference the instantiation link between the attribute mary!earns and its attribute class Employee!salary. The second frame shows that arbitrarily complex expressions are possible. The parenthesis has to be used to make the operator expressions unique. Though such complex expressions are rare in modeling, it is good to know that any object in O-Telos can be uniquely referenced in the frame syntax.

```
InstanceOf (mary!earns) -> (Employee!salary) with
 comment
    com1: "This is a comment to an instantiation link between attributes"
end
Attribute ((mary! earns) -> (Employee!salary))!com1 with
comment
com2: "This is a comment to the previous comment attribute"
end
```

Individual objects are denoted as nodes of the graph, instantiation, specialization and attribute relationships are represented as dotted, shaded, and labelled directed arcs between their source and destination components. Telos propositions have a temporal component: the belief time. The belief time of a proposition is not assigned by the user but by the system at transaction time of

an update (TELL or UNTELL).

3.4 Query language CBQL

In ConceptBase, queries are represented as classes, whose instances are the answer objects to the query. The system-internal object "QueryClass" may have so-called query classes as instances, which contain necessary and sufficient membership conditions for their instances. The syntax of query classes is a class definition with super classes, attributes, and a membership condition. The set of possible answers to a query is restricted to the set of common instances of all its super classes.

The following query computes all managers, which are bosses of an employee:

```
QueryClass AllBosses isA Manager with
constraint
    all_bosses_rule:
    $ exists e/Employee (e boss this) $
end
```

The predefined variable this in the constraint is identified with all solutions of the query class. Enter this query into the editor-window and press Ask (not Tell).The query will be evaluated by the server and after a few seconds the answer will appear both in the protocol and in the editor-window. In general for a given database each table is expressed as a class and each tuple is an instance of this class. This object-orientation has a lot of advantages. It helps us to view the database entries as real world objects and allows multiple values to be

entered for an attribute and their retrieval is also easy. There are many inbuilt queries .This visualization of database gives a better understanding and would be easier to provide the rules and constraints. Inheritance can be performed here and a class can serve as the attribute type of another class.

3.5 Query classes and constraints

ConceptBase regards query classes as ordinary classes with the only exception that class membership cannot be postulated (via a TELL) but is derived via the class membership constraint formulated for the query class. A consequence of this equal treatment is that a constraint formulated for an ordinary class can refer directly or indirectly to a query class, e.g.

```
Unit in Class with
Attribute
sub: Unit
end
BaseUnit in QueryClass isA Unit with
constraint
c1: $ not exists s/Unit!sub From (s,~this) $
end
SimpleUnit in Class isA Unit with
 constraint
c: $ forall s/SimpleUnit (s in BaseUnit) $
end
```

Here, the constraint in the class SimpleUnit refers to the query class BaseUnit

3.6 Query evaluation strategy

ConceptBase employs an SLDNF-style query evaluation method, i.e. query literals are evaluated top-down much like in standard Prolog. This is known to cause infinite loops for certain recursive rule sets. To overcome this, the SLDNF evaluator is augmented by a caching sub-system which detects recursive predicate calls and answers them from the cached results of a query rather than entering an infinite loop. This cache-based evaluation computes the fix point (explained in section 2.2) of a query provided that the overall rule set is stratified. Even more: also dynamically stratified rule sets are supported. Other than with the static stratification test, a violation is detected at run time of a query rather than at compile time. This makes it similar to the bottom-up evaluation method where the finite result is produced.

For a precise definition of stratification, we refer you to the literature on deductive databases. Consider the following rule:

```
forall p/Position (exists p1/Position (p moveTo p1) and not (p1 in Win))
  ==> (p in Win)
```

ConceptBase internally compiles such rules into a representation where Position, moveTo, and Win are predicate symbols:

```
forall p
   (exists p1 Position(p) and Position(p1) and
   move To(p,p1) and not Win(p1))
```

==> Win (p).

3.7 An example

A company has employees, some of them being managers. Employees have a name and a salary which may change from time to time. They are assigned to departments which are headed by managers. The boss of an employee can be derived from his department and the manager of that department. No employee is allowed to earn more money than his boss.
The model we want to create contains two levels: the class level containing the classes Employee, Manager and Department and the token level which contains instances of these 3 classes.

3.7.1 Class level

The first step is to create the three classes: Employee, Manager and Department.

Employee in Class

end

This is the declaration of the class Employee, which will contain every employee as instance. Employee is declared as instance of the system class Class, because it is on the class level of our example, i.e. it is intended to have instances. To add this object to the object base we have to press the Tell button. If no syntax error occurs and the semantic integrity of the object base isn't violated by this new object it will be added to the object base. The next class to

add is the class Manager. Managers are also employees, so the class Manager is declared as a specialization of Employee using the keyword isA:

Manager in Class isA Employee

end

The Department is defined as

Department in Class

end.

3.7.2 Defining attributes of classes

As mentioned in the description of the example-model, the employee-class has several attributes. To add them, we need to modify the Telos frame describing the class Employee.

Individual Employee in Class with

attribute

name: String;

salary: Integer;

dept: Department;

boss: Manager

end

Now you have added attributes to the class Employee. They are of the category attribute and their labels are: name, salary, dept, and boss. They establish "links" between the class Employee and the classes mentioned as "targets'". Department and Manager are user-defined classes, while String and Integer are built-in classes of ConceptBase.

Notice that these attributes are also available for the class Manager, because this class is a subclass of Employee (Explained in section 3.2.2 Specialization axiom). It is defined as

Individual Manager in Class isA Employee

end

Department is defined as:

Department in Class with

attribute

head: Manager

end

Here attribute "head" is of type Manager.

3.7.3 The token level

The company we model has four departments namely Production, Marketing, Administration and Research. Every employee working in the company belongs to a department. The employees will be listed later, apart from the managers of the departments:

Lloyd in Manager

end

Phil in Manager

end

Eleonore in Manager

end

Table 3.1 Department &Head of Employee class

Department	Head
Production	Lloyd
Marketing	Phil
Administration	Eleonore
Research	Albert

Albert in Manager

end

Next let us have a look at the department class,

Department in Class with

attribute

head: Manager

end

There is a link between Department and Manager of category attribute with label head at the class-level. Now we have to establish a link between Production and Lloyd of category head at the token-level. The label of this link must be a unique name for all links with the source object "Production". We choose head_of_Production as name. The resulting Telos frame is:

Production in Department with

Head

head_of_Production: Lloyd

end

Marketing in Department with

head

head_of_Marketing: Phil

end

Administration in Department with

head

head_of_Administration: Eleonore

end

Research in Department with

head

head_of_Research: Albert

end

Now the four managers have the following salaries

Lloyd in Manager with

salary

LloydsSalary: 100000

end

Table 3.2 Manager& Salary of Manager Class

Manager	Salary
Lloyd	100000
Phil	120000
Eleonore	20000
Albert	110000

Phil in Manager with

salary

PhilsSalary: 120000

end

Eleonore in Manager with

salary

EleonoresSalary: 20000

End

Albert in Manager with

salary

AlbertsSalary: 110000

end

Add the other employees to the object base as follows:

Michael in Employee with

dept

MichaelsDepartment: Production

salary

MichaelsSalary: 30000

end

Maria in Employee with

dept

Marias Department: Administration

salary

MariasSalary: 10000

end

Herbert in Employee with

dept

HerbertsDepartment: Marketing

salary

HerbertsSalary: 60000

end

Edward in Employee with

dept

EdwardsDepartment: Research

Salary

EdwardsSalary: 50000

end

3.7.4 Adding deductive rules

A deductive rule is of the format:

forall x1/c1 x2/c2 ……..xn/cn <Rule> = => lit (a1, ……..,am)

where <Rule> is a formula and the xi's are variables bound to the class ci , lit is a literal of type 1 or 3 (as given below) and the variables among ai's are exactly x1,…..xn.

The following literals may be used

1) x in c

The object x is an instance of class c.

2) c isA d

The object c is a specialization (subclass) of d.

3) x l y

The object x has an attribute to object y and this relationship is an instance of an attribute category with label 'l'. Structurally label l is an attribute of class x.

In order to avoid ambiguity, neither "in" and "isA" nor the logical nor the connectives "and" and "or" are allowed as attributes labels. The other set of literals that can be used for testing are given below and in a legal formula their parameters must be bound by one of the above mentioned literals.

4) x < y, x > y, x <= y, x >=y, x=y, x <>y

x and y must be instances of integer or real.

5) x == y

The objects x and y are the same. "and" and "or" are allowed as infix operators to connect sub formulas. Variables in formulas can be quantified by "for all x/c" or "exists x/c" where 'c' is a class and a range of 'x' is the set of all instances of the class 'c'.

Let us look at an example of deductive rule by defining the boss of an employee:

Employee with

rule

BossRule: $ forall e/Employee m/Manager

(exists d/Department

(e dept d) and (d head m))

==> (e boss m) $

end

The text of the formula must be enclosed in "$" and that this deductive rule is legal, because all variables appearing in the conclusion literal (e,m) are universal (forall) quantified. The logically equivalent formula is:

```
forall e/Employee m/Manager d/Department
    (e dept d) and (d head m)
    ==> (e boss m)
```

3.7.5 Adding integrity constraints:

The integrity constraint specifies that no Manager should earn less than 50000

```
Manager with
  constraint
    earnEnough: $ forall m/Manager x/Integer
                  (m salary x) ==> (x >= 50000) $
end
```

3.7.6 Defining queries

In ConceptBase queries are represented as classes, whose instances are the answer objects to the query. The system-internal object "QueryClass" may have so-called 'query classes' as instances, which contain necessary and sufficient membership conditions for their instances

The following query computes all managers, which are bosses of an employee:

```
QueryClass AllBosses isA Manager with
    constraint
```

```
all_bosses_srule:
    $ exists e/Employee (e boss this) $
end
```

The predefined variable 'this' in the constraint is identified with all solutions of the query class.We have seen a clear example of how to define class, attribute, and token, deductive rules, integrity constraints and queries in Conceptbase. Let us see in detail in fourth chapter how we perform similar things on bibtex database.

CHAPTER 4

EVALUATION AND OPTIMIZATION OF LOGIC QUERIES ON BIBTEX DATABASE USING CONCEPTBASE

In bibtex database bibliographic entries are classified into various categories: articles, book, in book, proceedings, in proceedings and so on. This database has to be transformed to a format that is compatible with conceptbase. In conceptbase answers are not given one tuple at a time like prolog. Third chapter explains on how to enter data and process queries using conceptbase. This chapter explains more on tables and queries. Tables are represented as classes and object-orientation concept inheritance is used. Let us see the tables we use and recursive queries that we solve in detail.

4.1 Tables

The three main tables are

(i)MASTER_ENTRY

(ii)PARENT_ID

(iii)RELATIONSHIP

4.1.1 MASTER_ENTRY table

Every instance of this table has the following attributes:

(i)Cite_key

(ii)Entry_type

(iii)Title

(iv)Author

(v)Publisher_Id

(vi)Reference

(vii)Relation

(viii)Number _of _pages

Cite_key is a string and it contains the cite_key of that instance. It is the primary key and it uniquely identifies each of the instances.

Entry_type is a string and it contains type details of that instance, whether it is an article, book, inbook and so on.

Title is a string and it contains the title of that instance.

Author is a string that contains the author of that instance.

Publisher_Id is a string that contains the id of its publisher.

Reference is of type PARENT_ID. The value is an instance of PARENT_ID that gives all the instances of MASTER_ENTRY that refers to this particular instance.

The Relation is of type RELATIONSHIP. The value is an instance of RELATIONSHIP that specifies the parent-id of each instance.

Number_of_pages is an integer that contains the number of pages of that instance.

4.1.2 The PARENT_ID table

PARENT_ID isA MASTER_ENTRY. In the sense it has the same

attribute as that of MASTER_ENTRY table. In other words it inherits MASTER_ENTRY table attributes.

4.1.3 The RELATIONSHIP table

It has an attribute Parent_Id of type PARENT_ID. It specifies the instance that directly refers to this entry.

4.2 An example

Consider an example where entries of MASTER_ENTRY table have cite_keys 10,11,12,13,14,15,16,17,18,19,20 where each of them are of different entry types like article, book, inbook along with the title, author and other information. Let us see how we can define this and enter values for considering a scenario where an article with cite_key 20 refers directly to 11, 12 and 15 and 15 refers to 13 and 13 refers to 10.

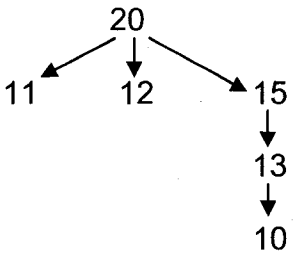

Figure 4. 1 Representation of example

First we need to define the three tables as a class. This is done by

MASTER_ENTRY in Class

end

PARENT_ID in Class isA MASTER_ENTRY

end

RELATIONSHIP in Class

end

As specified earlier PARENT_ID isA MASTER_ENTRY.

Now we define the attributes of MASTER_ENTRY and enter values for it.

Individual MASTER_ENTRY in Class with

attribute

Cite_key: String;

Entry_type: String;

Title: String;

Author: String;

Publisher_Id: String;

Reference: PARENT_ID;

Relation: RELATIONSHIP;

Number_of_pages: Integer

end

11 in MASTER_ENTRY

end

12 in MASTER_ENTRY

end

13 in MASTER_ENTRY

end

14 in MASTER_ENTRY

end

15 in MASTER_ENTRY

end

16 in MASTER_ENTRY

end

17 in MASTER_ENTRY

end

18 in MASTER_ENTRY

end

19 in MASTER_ENTRY

end

20 in MASTER_ENTRY

end

The values are entered as

10 in MASTER_ENTRY with

Cite_key

itscitekey:"A01"

Entry_type

itsentrytype:"ARTICLE"

Relation

Title:

itstitle: "A Comparision of Automatic Manual Zoning"

Author

itsfirstauthor: "John";

itssecondauthor: "Kim"

Publisher_Id

itspubid:"1025"

Number_of_pages

itsnumofpgs: 20

end

11 in MASTER_ENTRY with

Cite_key

itscitekey: "A02"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 11

Title

itstitle:"Information Processing and Management"

Author

itsauthor: "Johnson";

Publisher_Id

itspubid: "1026"

Number_of_pages

itsnumofpgs: 25

end

12 in MASTER_ENTRY with

Cite_key

itscitekey: "A03"

Entry_type

itsentrytype: "ARTICLE"

Relation

itsrelation: 12

Title

itstitle: "Information retrieval as statistical translation'"

Author

itsauthor: "Jackson"

Publisher_Id

itspubid: "1027"

Number_of_pages

itsnumofpgs: 30

end

13 in MASTER_ENTRY with

Cite_key

itscitekey: "A04"

Entry_type

itsentrytype: "ARTICLE"

Relation

itsrelation: 13

Title

itstitle: "Finding Acronyms and their Definitions"

Author

itsauthor: "David"

Publisher_Id

itspubid: "1028"

Number_of_pages

itsnumofpgs: 35

end

15 in MASTER_ENTRY with

Cite_key

itscitekey: "B01"

Entry_type

itsentrytype: "BOOK"

Relation

itsrelation: 15

Title

itstitle:"A Computational Morphology System'"

Author

itsauthor: "Bush"

Publisher_Id

itspubid: "1029"

Number_of_pages

itsnumofpgs: 40

end

The entries for PARENT_ID

15 in PARENT_ID

end

20 in PARENT_ID

end

13 in PARENT_ID

end

20 in PARENT_ID

end

13 in PARENT_ID with

Reference

itsreference: 15

end

15 in PARENT_ID with

Reference

itsreference: 20

end

The relationship table and its entries are given as

Individual RELATIONSHIP in Class with

attribute

Parent_Id:PARENT_ID

end

11 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

12 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

15 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

13 in RELATIONSHIP with

Parent_Id

itsparentid: 15

end

10 in RELATIONSHIP with

Parent_Id

itsparentid: 13

end

4.3 Queries

Query 1: To get the list of all instance that refer implicitly and explicitly to a particular instance. This is a recursive query. In this example we will find the list of all instances that refer to 10 implicitly and explicitly.

For this first we define the reference rule that will enter values for the Reference attribute of the MASTER_ENTRY table.

The rule is:

MASTER_ENTRY with

rule

Reference Rule: $ forall m/MASTER_ENTRY p/PARENT_ID (exists

r/RELATIONSHIP

(m Relation r) and (r Parent_Id p))

==> (m Reference p) $

end

This rule states that for all 'm' in MASTER_ENTRY and all 'p' in PARENT_ID, there exists a relation 'r' in the RELATIONSHIP and if that 'r' is the Relation attribute value of m and r's Parent_Id is p then p refers to m.

For example consider object 10 in MASTER_ENTRY and RELATIONSHIP:

10 in MASTER_ENTRY with

Cite_key

itscitekey: "A01"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 10

Title

itstitle:"A Comparision of Automatic Manual Zoning"

Author

itsfirstauthor:"John";

itssecondauthor:"Kim"

Publisher_Id

itspubid: "1025"

Number_of_pages

itsnumofpgs: 20

end

10 in RELATIONSHIP with

Parent_Id

itsparentid: 13

end

According to the rule if r=10 and p=13 then

10 Relation 10

10 Parent_Id 13

Therefore 10 Reference 13

Similarly,

11 Reference 20

12 Reference 20

13 Reference 15

15 Reference 20

This rule gets all the values for the Reference attribute of the MASTER_ENTRY table.In datalog this is expressed as

Reference (m, p):- Reference (m, p)

Reference (m, p):-Reference (m, r), Reference (r, p)

We can clearly see that it involves recursion. Now we have to find out the instances that refer to 10. The queryclass MetaReference contains all the answers to this query. It is defined as

QueryClass MetaReference isA PARENT_ID with

Constraint

References:

$ (10 Reference this) or

exists p/PARENT_ID

(p in MetaReference) and

(p Reference this)$

end

Here all the instances of PARENT_ID are analyzed one by one to see if they are in 10's Reference. For each of the value its Parent_Id 'p' is found and then added to MetaReference. Recursively check is made and all the objects that refer to 10

are retrieved and the stop is made when the value of Reference attribute is empty and no more PARENT_ID instances are left to be analyzed. That is reached at 20 in this example.

Objects of PARENT_ID =13, 15, 20

Start with 10

10 Reference 13 add 13 to MetaReference

13 Reference 15 add 15 to MetaReference

15 Reference 20 add 20 to MetaReference

20 Reference is emptystop the algorithm.

and no more PARENT_ID objects are left .

Then result is printed in the order of PARENT_ID instances as follows:

13 in MetaReference

end

15 in MetaReference

end

20 in MetaReference

end

Query 2: To get the list of all instances that a particular object refers to implicitly and explicitly. This is a recursive query. In this example let us find out the list of instances that 20 refer to.

Here we define an attribute called Ref in PARENT_ID. And the rule is

PARENT_ID with

Attribute

Ref: MASTER_ENTRY

Rule

isref: $ forall p/PARENT_ID m/MASTER_ENTRY

(m Reference p)

==>

(p Ref m) $

end

This rule states that for all 'p' in PARENT_ID and all 'm' in MASTER_ENTRY, if the value of Reference attribute of m is p then it implies that p refers m. These values are stored in attribute Ref of PARENT_ID.

From the previous query we know that

10 Reference 13

11 Reference 20

12 Reference 20

13 Reference 15

15 Reference 20

Considering the rule defined now

13 Ref 10

20 Ref 11

20 Ref 12

15 Ref 13

20 Ref 15

In datalog this is expressed as

Ref (p, m):- Reference (m, p)

Ref (p, m):- Ref (p, m)

Ref (p1, p2):- Ref (p1, m), Ref (m, p2)

We can clearly see that recursion is involved. Now we write the queryclass Metaref that will contain all the objects that 20 refer to implicitly and explicitly.

QueryClass Metaref isA MASTER_ENTRY with

constraint

refs:

$ (20 Ref this) or

exists m/MASTER_ENTRY

(m in Metaref) and

(m Ref this)$

end

Here all the instances of MASTER_ENTRY are analyzed one by one to find if they are in the Ref attribute of 20. If so they are added to Metaref and recursively the search continues till Ref is empty and no more MASTER_ENTRY instances are left to analyze.

Objects of MASTER_ENTRY =10, 11,12,13,14, 15, 16, 17, 18, 19, 20

Start with 20

20 Ref 11 add 11 to Metaref

11 Ref empty stop this loop

20 Ref 12 add 12 to Metaref

12 Ref emptystop this loop

20 Ref 15 add 15 to Metaref

15 Ref 13 add 13 to Metaref

13 Ref 10add 10 to MetaRef

10 Ref empty and stop the algorithm

no more objects of MASTER_ENTRY are left to analyze

The answer is printed in the order of MASTER_ENTRY objects as

10 in Metaref

end

11 in Metaref

end

12 in Metaref

end

13 in Metaref

end

15 in Metaref

end

Query 3: To find all the authors of a particular instance. This is not recursive. ConceptBase allows an attribute to have more than one instance. This query tries to get all the values of Author attribute.

find_attribute_values[MASTER_ENTRY!Author/cat, 10/objname]

In datalog it is expressed as

Author (10, John)

Author (10, Kim)

q: Author (10, x)

This is not a recursive query. Here all the results that match the x value from the set of EDB's are displayed as the answer. The two parameters to be entered are category and object name. The category is MASTER_ENTRY!Author and the object name is 10. The answer is displayed as

Answer:

"John" in find_attribute_values[10/objname,MASTER_ENTRY!Author/cat]

end

"Kim" in find_attribute_values [10/objname, MASTER_ENTRY!Author/cat]

end

Query 4: To find all instances to which other instances refer to. This is a non-recursive query. It gets all the instances that are being referenced by the instances of the MASTER_ENTRY table. The query may be expressed in datalog as

For all 'm' of the MASTER_ENTRY table

q: Reference (m, X)

Here all the values that match X from the set of EDB's is printed as the answer. The query class AllParentIds contains all the answers to this query. In conceptbase it is defined as follows:

QueryClass AllParentIds isA PARENT_ID with

Constraint

all_parentsrule:

$ exists m/MASTER_ENTRY (m Reference this) $

end

Answer:

13 in AllParentIds

end

15 in AllParentIds

end

20 in AllParentIds

end

Query 5: To get all instances whose entry type is "ARTICLE" from the MASTER_ENTRY table. This is a non-recursive query. In this query the entry type of all instances of MASTER_ENTRY table is checked .The ones that have "ARTICLE" as their entry type is printed as the answer.

In datalog this is expressed as

q: Entry_type(X, "ARTICLE") for all objects 'm' in MASTER_ENTRY the values that match X are printed as the answer. In conceptbase it is defined as follows:

QueryClass ArQuery isA MASTER_ENTRY with

constraint

type:

$ (this Entry_type "ARTICLE") $

end

Answer:

10 in ArQuery

end

11 in ArQuery

end

12 in ArQuery

end

13 in ArQuery

end

Query 6: To find the number of instances of a particular class. This is a non-recursive query. Here we try to find the how many instances a particular class contains. In datalog it is expressed as

q: count (class name)

If we want to find the number of instances of MASTER_ENTRY class then in conceptbase it is defines as:

COUNT[MASTER_ENTRY/class]

Answer:

10 in COUNT [MASTER_ENTRY/class]

end

Query7: To find all classes the given object belongs to. This is non-recursive. In datalog it is expressed as

q: find_classes(objectname)

To find all classes that object '10' belongs to in conceptbase we do the following:

find_classes[10/objname]

Answer:

Proposition in find_classes[10/objname]

end

MASTER_ENTRY in find_classes[10/objname]

end

PARENT_ID in find_classes[10/objname]

end

RELATIONSHIP in find_classes[10/objname]

end

Integer in find_classes[10/objname]

end

Individual in find_classes[10/objname]

end

Query 8: To retrieve a particular instance. This is a non-recursive query. It tries to retrieve a particular object. In datalog it is expressed as

get_object(objectname)

To retrieve object 10 we do the following in conceptbase:

get_object[10/objname]

Answer:

Individual 10 in MASTER_ENTRY,PARENT_ID,RELATIONSHIP,Integer with

Parent_Id,attribute

    itsparentid: 13

Cite_key,attribute

itscitekey: "A01"

Entry_type,attribute

itsentrytype: "ARTICLE"

Relation,attribute

itsrelation: 10

Reference,attribute

Itsreference: 13

Author,attribute

itsfirstauthor: "John";

itssecondauthor: "Kim"

Publisher_Id,attribute

itspubid: "1025"

Number_of_pages,attribute

itsnumofpgs: 20

end

Query 9: To check whether a given object exists. In datalog it can be expressed as

exists (objectname)

To find whether object 10 exists, in conceptbase we do the following:

exists [10/objname]

Answer: yes

Query 10: To find all the instances of a given class. In this query we try to find all the instances of a given class. This is not recursive and in datalog it

is expressed as

find_instances (class name)

To find all the instances of MASTER_ENTRY table we do the following in conceptbase:

find_instances[MASTER_ENTRY/class]

Answer:

10 in find_instances[MASTER_ENTRY/class]

end

11 in find_instances[MASTER_ENTRY/class]

end

12 in find_instances[MASTER_ENTRY/class]

end

13 in find_instances[MASTER_ENTRY/class]

end

14 in find_instances[MASTER_ENTRY/class]

end

15 in find_instances[MASTER_ENTRY/class]

end

16 in find_instances[MASTER_ENTRY/class]

end

17 in find_instances[MASTER_ENTRY/class]

end

18 in find_instances[MASTER_ENTRY/class]

end

19 in find_instances[MASTER_ENTRY/class]

end

20 in find_instances[MASTER_ENTRY/class]

end

Query 11: To find the number of values a given attribute has for a given object. In this query we try to find out the number of values associated with a particular attribute of a particular object. In datalog it is expressed as

COUNT_Attribute (objectname,attributecategory).So to find the number of values for the "Author" attribute of object 10 in conceptbase ,it is given as:

COUNT_Attribute [10/objname, MASTER_ENTRY!Author/attrcat]

Answer:

2 in COUNT_Attribute[10/objname,MASTER_ENTRY!Author/attrcat]

end

Query 12: To find all the work of a given author. This query is non-recursive. It tries to find all the articles, books, proceeding etc...of an author. In datalog it is expressed as

Author (10, John)

q: Author(X, John)

Here we get all the articles, books etc written by John. In conceptbase it is represented as:

QueryClass AuthQuery isA MASTER_ENTRY with

constraint

type:

$ (this Author “John") $

end

Answer:

10 in AuthQuery

end

Query 13: To find all the work of a given publisher. This query is non-recursive. Here we try to find all the articles, books, etc...of a given publisher. In datalog it is expressed as

Publisher_Id (10, 1025)

q: Publisher_Id(X,1025)

Here we get all the articles, books etc published by a publisher with publisher_id 1025. In conceptbase it is represented as:

QueryClass PubQuery isA MASTER_ENTRY with

constraint type:

$ (this Publisher_Id “1025") $

end

Answer:

10 in PubQuery

end

Query 14: To find all works published by a particular author and publisher. This is non-recursive and here we have two constraints the article, books etc should have the given author and publisher. In datalog it is expressed as

Author (10, John)

Publisher_Id (10, 1025)

q: Author(X, John) , Publisher_Id (X,1025)

In conceptbase to get the answer to this query we do the following:

QueryClass PubAuthQuery isA MASTER_ENTRY with

constraint

type:

$ (this Author "John") and (this Publisher_Id "1025") $

end

Answer:

10 in PubAuthQuery

end.

Query 15: To find the number of pages of a given article, book, proceedings etc. This is non-recursive. In datalog it is expressed as

find_attribute_values (objectname, category)

In conceptbase we define it as:

find_attribute_values [10/objname, MASTER_ENTRY!Number_of_pages/cat]

Answer

20 in find_attribute_values[10/objname,MASTER_ENTRY!Number_of_pages/cat]

end

Query 16: To find the publisher_id of a particular object. This query is non-recursive. In datalog it is expressed as:

find_attribute_values (category, object name)

In conceptbase it is defined as:

find_attribute_values[MASTER_ENTRY!Publisher_Id/cat,10/objname]

Answer:

"1025" in find_attribute_values[MASTER_ENTRY!Publisher_Id/cat,10/objname]

end

Query 17: To find the title of a given object. It is non–recursive. In datalog it is expressed as

find_attribute_values (object name, category). In conceptbase it is defined as:

find_attribute_values[10/objname,MASTER_ENTRY!Title/cat]

Answer

"A Comparision of Automatic Manual Zoning" in

find_attribute_values[10/objname,MASTER_ENTRY!Title/cat]

end

Now consider the following example

1)Kazem Taghva, Julie Borsack, Steven Lumos, and Allen Condit. **A Comparison of Automatic and Manual Zoning: An Information Retrieval Prospective**. *Int. Journal on Document Analysis and Recognition*, 6(4):230-235, April 2004.

2)W. B. Croft, S. Harding, K. Taghva, and J. Borsack. **An evaluation of information retrieval accuracy with simulated OCR output**. In Proc. 3rd Symposium on Document Analysis and Information Retrieval, pages 115-126, Las Vegas, NV, April 1994.

3)D. Harman. **Information Retrieval, Data Structures and Algorithms**, chapter Ranking Algorithms, pages 363-392.Prentice Hall, Englewood Cli®s, NJ 07632, 1992.

4)Kazem Taghva, Julie Borsack, and Allen Condit. **Effects of OCR Errors on Ranking and Feedback Using the Vector Space Model**. *Inf. Proc. and Management*, 32(3):317-327, 1996.

5)Kazem Taghva, Julie Borsack, and Allen Condit. **An Expert System for Automatically Correcting OCR Output**. In *Proc. IS&T/SPIE 1994 Intl. Symp. on Electronic Imaging Science and Technology*, pages 270-278, San Jose, CA, February 1994.

6)Kazem Taghva, Julie Borsack, Allen Condit, and Srinivas Erva. **The Effects of Noisy Data on Text Retrieval**. *J. American Soc. for Inf. Sci.*, 45(1):50-58, January 1994.

Here paper 1 refers to 2, 3, 4, 5 and 6. Paper 2 refers to 3 and 4. Paper 3 refers to paper 4. This expresses the same relation as

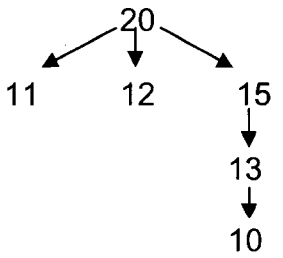

Figure 4. 2 Representation of Example

Where

20 - A Comparison of Automatic and Manual Zoning: An Information Retrieval Prospective

11 - An evaluation of information retrieval accuracy with simulated OCR output

12- Information Retrieval, Data Structures and Algorithms

15- Effects of OCR Errors on Ranking and Feedback Using the Vector Space Model

13- An Expert System for Automatically Correcting OCR Output

10- The Effects of Noisy Data on Text Retrieval

With this relationship let us now enter the details in the MASTER_ENTRY table, PARENT_ID table and RELATIONSHIP table

10 in MASTER_ENTRY with

Cite_key

itscitekey: "A01"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 10

Title

itstitle:" The Effects of Noisy Data on Text Retrieval "

Author

itsfirstauthor:"Kazem Taghva ";

itssecondauthor:"Julie Borsack";

itsthirdauthor:"Allen Condit";

itsfourthauthor:"Srinivas Erva"

Number_of_pages

itsnumofpgs: 8

end

11 in MASTER_ENTRY with

Cite_key

itscitekey: "A02"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 11

Title

itstitle:" An evaluation of information retrieval accuracy with simulated OCR output."

Author

itsfirstauthor:"W.B.Croft";

itssecondauthor:"S.Harding";

itsthirdauthor:"K.Taghva";

itsfourthauthor:"J.Borsack"

Number_of_pages

itsnumofpgs: 11

end

12 in MASTER_ENTRY with

Cite_key

itscitekey: "B01"

Entry_type

itsentrytype:"BOOK"

Relation

itsrelation: 12

Title

itstitle:" Information Retrieval, Data Structures and Algorithms "

Author

itsauthor:"D.Harman ";

Number_of_pages

itsnumofpgs: 29

end

13 in MASTER_ENTRY with

Cite_key

itscitekey: "A03"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 13

Title

itstitle:" An Expert System for Automatically Correcting OCR Output "

Author

itsfirstauthor:” Kazem Taghva”;

itssecondauthor:" Julie Borsack ";

itsthirdauthor:” Allen Condit”

Number_of_pages

itsnumofpgs: 11

end

15 in MASTER_ENTRY with

Cite_key

itscitekey: "A04"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 15

Title

itstitle:" An Expert System for Automatically Correcting OCR Output "

Author

itsfirstauthor:” Kazem Taghva”;

itssecondauthor:" Julie Borsack ";

itsthirdauthor:” Allen Condit”

Number_of_pages

itsnumofpgs: 8

end

20 in MASTER_ENTRY with

Cite_key

itscitekey:"A05"

Entry_type

itsentrytype:"ARTICLE"

Relation

itsrelation: 20

Title

itstitle:" A Comparison of Automatic and Manual Zoning: An Information Retrieval Prospective "

Author

itsfirstauthor:" Kazem Taghva";

itssecondauthor:" Julie Borsack ";

itsthirdauthor:" Steven Lumos" ;

itsfourthauthor: "Allen Condit"

Number_of_pages

itsnumofpgs: 5

end

The PARENT_ID entries are

20 in PARENT_ID

end

13 in PARENT_ID with

Reference

itsreference: 15

end

15 in PARENT_ID with

Reference

itsreference: 20

end

The RELATIONSHIP entries are

10 in RELATIONSHIP with

Parent_Id

itsparentid: 13

end

11 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

12 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

13 in RELATIONSHIP with

Parent_Id

itsparentid: 15

end

15 in RELATIONSHIP with

Parent_Id

itsparentid: 20

end

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Recursive queries are processed and implemented in Datalog fashion using ConceptBase. Answers to these queries are not got one-tuple at a time and standard query processing strategies are involved. A detailed description of bibliographic database is provided. Three tables (i) MASTER_ENTRY (ii) PARENT_ID (iii) RELATIONSHIP are defined. ConceptBase is object oriented and so relationships between these tables are defined carefully to be compatible with it. An example that shows in reality the same relationship defined between these tables is explained and how data has to be entered in ConceptBase for this example is shown for clear understanding.

Deductive databases are analyzed in chapter 1 and general evaluation and optimization techniques for recursive queries are explained in chapter 2. All basics of ConceptBase are explained in chapter 3 and how to handle recursive queries using ConceptBase is given in detail. Recursive queries and scenarios where recursion occurs are dealt in chapter 4. A set of queries and their equivalent datalog expressions and the rules to define them are given. Object oriented concept inheritance is used; PARENT_ID table is inherited from

MASTER_ENTRY table. ConceptBase allows a particular attribute to take more than one instance. Thus if an instance has more than one author we can enter in the values and retrieve all of them unlike general databases ,where we have to create a separate field if a particular attribute takes more than one value. This feature is depicted in Query 3, section 4.3. In contrast to this, given a particular author or publisher all their works are displayed in Query 12 & Query 13, section 4.3. Thus processing of recursive and other queries in an object-oriented environment is studied and implemented.

5.2 Future work

In this thesis, we have given an overview of logic queries and their implementation. We have shown how a bottom-up approach computes a recursive query using a concrete example with applications for bibliographic databases. Future work will focus on experimental analysis to compare the time complexity of the ConceptBase approach with other approaches such as XML query processing using XQUERY and XSLT.

REFERENCES

[1]Francous Bancilhon, Raghu Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies.

[2] Juliana Freire, Using Logic Programming to Efficiently Evaluate Recursive Queries

[3]Notes on Datalog - http://infolab.stanford.edu/~ullman/dscb/pslides/dlog.ppt

[4]Raghu Ramakrishnan, Jeffrey .D. Ullman, A survey of Research on Deductive Database Systems

[5] S Krishna, Introduction to Database and Knowledge-base systems, World Scientific Series in Computer Science - Volume 28

[6] Jeffery .D. Ullman, Principles of Database & Knowledge-Base Systems Vol. 1: Classical Database Systems

[7]Jeffery .D. Ullman, Principles of Database and Knowledge-Base Systems Volume II: The New Technologies

[8]Catriel Beeri, Raghu Ramakrishna, On the Power of Magic

[9]Seppo Sippu, Eljas Soisalon- Soininen, An analysis of magic sets and related optimization strategies for logic queries

[10]ConceptBase site- http://dbis.rwth-aachen.de/CBdoc/

[11]ConceptBase tutorial - http://dbis.rwth-aachen.de/CBdoc/tutorial/

[12]ConceptBase user manual - http://dbis.rwth-aachen.de/CBdoc/userManual/

VITA

Graduate College
University of Nevada, Las Vegas

Jayalakshmi Jeyaraman

Address:
1780 Creekside Drive Apt #1023
Folsom, CA 95630

Degree:

- Bachelor of Engineering, Computer Science, 2006 Anna University, India

Thesis Title: Implementation of Recursive Queries for Information Systems

Thesis Examination Committee:

- Chairperson, Dr. Kazem Taghva, Ph.D.
- Committee Member, Dr. Ajoy.K.Datta,, Ph.D.
- Committee Member, Dr. Wolfgang Bein, Ph.D.
- Graduate College Representative, Dr Emma Regentova, Ph.D.