Integrity Constraint Reformulation for Efficient Validation

Xiaolei Qian
Department of Computer Science, Stanford University, CA 94305
And Kestrel Institute

Douglas R. Smith
Kestrel Institute, 1801 Page Mill Road, Palo Alto, CA 94304

Abstract

Constraint validation has been difficult to implement efficiently. The major reason for this difficulty lies in the state-dependent nature of integrity constraints and the requirement of both high-level specification and efficient runtime enforcement. In this paper, we propose a constraint reformulation approach to efficient constraint validation. We also demonstrate how this knowledge-based constraint reformulation can be naturally accomplished in the general framework of problem reformulation with the technique of antecedent derivation. We formalize the reformulation of an integrity constraint as a tree-search process where the search space is the set of all semantic-equivalent alternatives of the original constraint. We also develop control strategies and meta-level rules for carrying out the search efficiently. The major contribution of this work is a new promising approach to efficient constraint validation and a general framework to accomplish it.

1. Introduction

Constraint validation, an essential feature of any database systems, is the process of guaranteeing and maintaining a set of semantic invariants across database state transitions. This process has been very difficult to implement efficiently[1,4]. The major reason for this difficulty lies in the state-dependent nature of integrity constraints and the requirement of both high-level specification and efficient runtime enforcement. Research on constraint validation has concentrated on deriving efficient algorithms from the syntactic structure of constraint specification[6,8,10,12,13,21]. No knowledge of the changing world and the actual implementation of the database has been used to obtain such algorithms because they are derived once for all possible situations. These approaches neglect the fact that integrity constraints closely relate to specific database states in the sense that they have to be validated against specific states. Hence only sub-optimal performance can be expected.

Bernstein, Blaustein, and Clarke gave an improved scheme in [2] in which some very limited primitive information (in forms of auxiliary aggregate data) is maintained in order to improve the performance of constraint enforcement. However, they only considered a small class of database integrity constraints involving arithmetic comparison operators. Paige applied the finite differencing technique to constraint validation in [14]. Although maintaining auxiliary information is very effective in reducing expensive recomputations to incremental updates, blindly applying it without considering the usage pattern of database sometimes leads to more costly operations. Neither of these approaches provides control over the usage of auxiliary information.

All approaches mentioned above take the integrity constraint specification as it is given by the user. In [16] we proposed a different approach to the efficient validation of integrity constraints, which seeks to exploit knowledge about the application domain and database implementation to reformulate user-specified constraints into ones which are syntactically different but semantically equivalent in the sense that they enforce the same condition given the application semantics, and which are cheaper to implement given the existing database configuration. Such a knowledge-based approach provides great potential for efficient implementation because: logically equivalent constraint specifications can have very different computational characteristics; by exploring knowledge about application semantics and database configurations, constraints are specialized to the current run time environment with more optimization opportunities; and since the process of constraint reformulation is automated, it is easy to adapt to change in application semantics and database organization.

The basic idea behind our approach is similar in spirit to the one proposed by Hammer[5] and King[9] for knowledge-based query optimization, in the sense that we are also looking for optimization by semantic transformation. However there are important differ-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

ences in both the nature of the problem to be solved and the effectiveness of the solution. First, query optimization is performed far more frequently than constraint reformulation. King put severe restrictions on the data model and the type of constraints allowed in order to avoid the overhead of general deduction. In the approach of Hammer, the search space is not characterized, the cost model of both query improvement and search efficiency is intuitively described, and the control mechanism is incomplete. It is unclear how effective the control mechanism is against search space explosion and how semantic reasoning interacts with heuristic search. Finally, Hammer only supports restricted forms of logically equivalent transformations because his knowledge reprsentation is not suitable for deductive use.

We demonstrate how knowledge-based constraint reformulation can be accomplished in the general framework of problem reformulation with the technique of antecedent derivation, which serves as a search space generator of all alternative reformulations. We develop a cost model as the criterion for selecting the most efficient reformulation. Finally we propose a set of strategies for carrying out the search for the best reformulation efficiently. The whole reformulation process of a constraint is formalized as a tree-search process where the set of all valid reformulations of the original constraint forms a tree.

We restrict our attention in this paper to the reformulation of single constraints with respect to a set of knowledge in relational databases, although the techniques presented provide basis for and can be extended to the reformulation of a set of constraints. The integrity constraints are specified in first-order logic augmented with set-theoretic operators and reduction (aggregation) operators. Knowledge about the application domain and database configuration is also represented as logical assertions. The paper is organized as follows. Section 2 gives a formal specification of the knowledge-based constraint reformulation together with a classification of the knowledge used. In Section 3 we briefly describe a deductive system called RAINBOW which serves as the inference engine. We then present, in Section 4, a cost function to measure the preference of reformulations. Finally our search control strategy is presented in Section 5, which consists of a set of task ordering and pruning heuristics. We illustrate concepts by using the example database and constraint below.

DATABASE SCHEMA

EMP(EName, EDept, Sal)
DEPT(DName, Bgt, Chairman, MaxSal)
PROJ(PName, PDept, PMgr)
ASSIGN(AEmp, AProj, Percent)
SKILL(SEmp, SName)

CONSTRAINT

$(\forall e \in EMP)(\forall a \in ASSIGN)$
$(\forall p \in PROJ)(\forall d \in DEPT)$
$[EName(e) = AEmp(a) \wedge$
$AProj(a) = PName(p) \wedge$
$PDept(p) = DName(d)$
$\Rightarrow Sal(e) \leq Bgt(d)]$

2. Knowledge-based Constraint Reformulation

A database is a collection of relations, each relation is a set of tuples. The structure of the database is characterized by its schema, which specifies all the relations, attributes, and domains on which the values of attributes are defined. Relational databases do not explicitly support the specification of inter-relation relationships. They are made possible by matching domains of attributes in the relations involved in the relationships. We call such attributes *connection attributes.* An integrity constraint is an abstraction of a validity condition that data in the database must obey. The set of integrity constraints together serve as the correctness criteria for valid database states.

The knowledge-based reformulation of constraints can be formally specified as: given a knowledge base K of a set of assertions and a constraint P, we want to find an assertion P' such that (1) K together with P' implies P, i.e., $K \wedge P' \Rightarrow P$; and (2) P' is both computationally cheaper than P and semantically as weak as possible. We call P' an antecedent of P (relative to K). With the assumption that all the assertions in the knowledge base K are valid in the current database state, checking a constraint P' will be more cost-effective than enforcing the original constraint P and at the same time guarantee that the database invariant P is properly maintained.

A wide range of knowledge can be explored in constraint reformulation[22]. The knowledge falls into several categories, from domain-specific application semantics to specific implementation techniques used in the database configuration.

- Application semantics, such as the cardinality of relationships[3], other integrity constraints that are already validated, and the current state of the application.
- Database structures, such as the available logical structures supported by the database, virtual and derived information, and existing bindings.
- Physical organization and access paths, such as the materialized links and indexes, physical clustering and locality.
- Database utilization through monitoring, such as the update frequency of relations and usage frequency of access paths.

- Performance and cost information, such as the relation and image sizes, blocking factors, and the time to access a particular index.
- Algebraic properties of operations, such as the existence of inverse of a function or mapping, and estimated set size after set-former operation.

The process of constraint reformulation occurs at "compile time", in the sense that efficient checking code is generated from the reformulated constraint to actually enforce it at run time. However, under two situations the reformulation process has to be retracted, that is, the reformulation of P into P' becomes invalid: (1) when some knowledge in the knowlege base, which has been used in the reformulation, has become invalid due to a database state transition; (2) when an implication is used in the reformulation such that $P' \Rightarrow P$, but after an intended state transition, P' becomes false and P remains true. In both cases the reformulation should be redone instead of invalidating the state transition. By carefully choosing the knowledge to be used we can minimize the cost of multiple reformulation. In Section 5 we develop control strategies on the use of temporal or implication knowledge.

3. Antecedent Derivation

In this section we briefly describe a formal deductive system called RAINBOW developed at Kestrel Institute[18,19]. RAINBOW is a system for deriving antecedents. Given a goal G and assertion H it tries to find a formula P, called a *derived antecedent*, such that $H \wedge P \Rightarrow G$. The deduction process has two stages. In the first stage reduction rules are repeatedly applied to goals reducing them to subgoals. A primitive rule is applied whenever possible. The result of this reduction process is a goal tree in which (1) nodes represent goals/subgoals, (2) arcs represent reduction rule applications, and (3) leaf nodes represent goals to which primitive rules have been applied. The second stage involves the bottom-up composition of antecedents for subgoals into an antecedent for the parent goal.

If $\{x_1, ..., x_n\}$ are the free variables in G, then an $\{x_1, ..., x_i\}$-*antecedent* of G is a formula P whose free variables $\{x_1, ..., x_i\} \subseteq \{x_1, ..., x_n\}$ such that

$$H \Rightarrow \forall x_1 ... \forall x_i [P \Rightarrow \forall x_{i+1} ... \forall x_n G]$$

P is a *weakest* $\{x_1, ..., x_i\}$-*antecedent* if

$$H \Rightarrow \forall x_1 ... \forall x_i [P \equiv \forall x_{i+1} ... \forall x_n G]$$

is valid. Given a goal G with a set of free variables X, RAINBOW looks for all possible formulas which are an X'-*antecedent* of G where $X' \subseteq X$.

There are 10 reduction rules, 3 primitive rules, and two composition methods in the system. Only the portion used in our examples is presented here. All formulas in the rest of the paper are assumed to be implicitly quantified and all free variables are treated as constants. In presenting the reduction rules we use the notation $G; H$ as an abbreviation of the formula $h_1 \wedge h_2 \wedge ... \wedge h_k \Rightarrow G$ where $H = \{h_1, h_2, ..., h_k\}$. A complete description may be found in [18].

R3. *Reduction of Conjunctive Hypotheses.* If the goal is $G; H \cup \{B \wedge C\}$, then generate subgoals $G; H \cup \{B\}$ and $G; H \cup \{C\}$. If these subgoals return antecedents A_1 and A_2, then return the disjunctive composition $A_1 \vee A_2$ as the antecedent of the goal.

R5. *Application of an Equivalence Formula.* If the goal is $G; H$ and $A \equiv G$ is a known theorem or an assertion in H then generate subgoal $A; H$.

R7. *Forward Inference from an Assertion.* If the goal is $G; H$, $A \Rightarrow B$ or $A \equiv B$ is a known theorem or assertion in H, and A is an assertion in H, then generate subgoal $G; H \cup \{B\}$.

R8. *Goal/Assertion Duality Rules.* (a) If the goal has the form $\neg A \vee B; H$ then generate subgoal $B; H \cup \{A\}$. (b) If the goal is $B; H$ and $A \in H$ then generate subgoal $\neg A \vee B; H - \{A\}$.

R9. *Substitution of Equal Terms.* (a) If the goal has the form $G(r); H$ and $r = s$ is an assertion in H or a known theorem then generate subgoal $G(s); H$. (b) If one assertion has the form $h(r)$ and $r = s$ is another assertion or a known theorem, then generate subgoal $G; H \cup h(s)$.

P1. *Primitive Rule.* If the goal is $G; H$, we seek an $\{x_1, ..., x_n\}$-*antecedent*, G and H' depend only on the variables $x_1, ..., x_n$ where H' has the form $\wedge_{j=1}^{m} h_{i_j}$, and $\{h_{i_j}\}_{j=1,...,m} \subseteq H$, then generate antecedent $H' \Rightarrow G$.

Figure 1 gives a set of knowledge and theorems about our example database. Now suppose that we want to derive an $\{e\}$-*antecedent* of our example constraint. A goal tree representing a formal derivation of the antecedent $Sal(e) \leq Bgt(EMPIN(e))$ is shown in Figure 2. The arcs of the goal tree are annotated with the names of the rules and known theorems or assertions used. The leaves of the goal tree are annotated with the primitive rules used. Figure 3 shows the derivation of a $\{d\}$-*antecedent*: $MaxSal(d) \leq Bgt(d)$.

There are several interesting features of this example that are worth mentioning. First the deductive problem of antecedent derivation matches perfectly to our constraint reformulation specification in Section 2. The set of assertions plays the role of a knowledge base and the goal is the constraint we want to reformulate. RAINBOW provides us with a framework of systematically generating the space of antecedents. The correct-

ness of the alternative constraints generated by RAINBOW is straightforward from the correctness of reduction rules.

Assertions:

h1. $MaxSal(d) = max\{Sal(e): e \in HASEMP(d)\}$

h2. $(DName(d_1) = DName(d_2)) \Rightarrow (d_1 = d_2)$

h3. $EDept(e) = DName(EMPIN(e))$

h4. $(e \in HASEMP(d)) \equiv (EMPIN(e) = d)$

h5. $(EName(e) = AEmp(a) \wedge AProj(a) = PName(p) \wedge PDept(p) = DName(d)) \Rightarrow (EDept(e) = DName(d))$

h6. $(EName(e) = s) \equiv (e = IEName(s))$

h7. $(PName(p) = s) \equiv (p = IPName(s))$

h8. $(DName(d) = s) \equiv (d = IDName(s))$

Theorems:

t1. $(P \Rightarrow Q) \equiv (\neg P \vee Q)$

t2. $((e \in S) \wedge P \Rightarrow (F(e) \leq C)) \equiv ((max\{F(e): e \in S \wedge P\}) \leq C)$

Figure 1: Assertions and theorems

Derived Assertions:

d1. $EName(e) = AEmp(a) \wedge AProj(a) = PName(p) \wedge PDept(p) = DName(d)$

d2. $EDept(e) = DName(d)$

d3. $DName(EMPIN(e)) = DName(d)$

d4. $EMPIN(e) = d$

Goal:

G1. $(EName(e) = AEmp(a) \wedge AProj(a) = PName(p) \wedge PDept(p) = DName(d)) \Rightarrow (Sal(e) \leq Bgt(d)); H$
↓ R5+t1, R8(a)

G2. $Sal(e) \leq Bgt(d); H \cup \{d1\}$
↓ R7+h5+d1

G3. $Sal(e) \leq Bgt(d); H \cup \{d1, d2\}$
↓ R9(b)+h3+d2

G4. $Sal(e) \leq Bgt(d); H \cup \{d1, ..., d3\}$
↓ R7+h2+d3

G5. $Sal(e) \leq Bgt(d); H \cup \{d1, ..., d4\}$
↓ R9(a)+d4

G6. $Sal(e) \leq Bgt(EMPIN(e)); H \cup \{d1, ..., d4\}$
↓ P1

Figure 2: An $\{e\}$-*antecedent* of the goal

Secondly, the assertions in the example represent typical knowledge about our database application. Assertion h2 expresses the key constraint of the *DEPT* relation. Assertions h3 and h4 say that there is a many-to-one mapping called *EMPIN* from *EMP* to *DEPT* connecting each employee to his department, and its inverse is a one-to-many mapping *HASEMP*. Assertions h6, h7, and h8 specify the three indexes on attributes *EName*, *PName*, and *DName*. By associating cost information with these functional mappings we are able to incorporate physical organization knowledge into our deductive framework.

Both derivations use the assertion h5, which says that (currently) employees only involve in projects in their own departments. With this piece of information our constraint gets greatly simplified because we can compare the employee's salary directly with the budget of his department, without going through all of his projects. The two reasons we mentioned before for retracting reformulation are both possible with the introduction of h5 in our derivation. The database may evolve into a state where it is no longer true that each employee only works for his own department. It may very well be the case that an employee is joining a multi-department project. This makes h5 not valid and checking P' is not enough for ensuring the validity of P. Also by using the reformulated constraint instead of the original one, we are enforcing a stronger condition than necessary, a condition that says each employee must earn no more than the budget of his department — even if he is not involved in any project (in his department). Given that it is usually the case that every employee works for at least one project, it may still be beneficial to enforce this stronger constraint.

Derived Assertions:

d5. $e \in HASEMP(d)$

Goal:

G5. $Sal(e) \leq Bgt(d); H \cup \{d1, ..., d4\}$
↓ R7+h4+d4

G7. $Sal(e) \leq Bgt(d); H \cup \{d1, ..., d5\}$
↓ R8(b)+d5, t1

G8. $(e \in HASEMP(d)) \Rightarrow Sal(e) \leq Bgt(d); H \cup \{d1, ..., d5\}$
↓ R5+t2

G9. $max\{Sal(e): e \in HASEMP(d)\} \leq Bgt(d); H \cup \{d1, ..., d5\}$
↓ R9(a)+h1

G10. $MaxSal(d) \leq Bgt(d); H \cup \{d1, ..., d5\}$
↓ P1

Figure 3: A $\{d\}$-*antecedent* of the goal

In the second derivation shown in Figure 3, the 6th reduction step from G7 to G8 introduces the inverse mapping *HASEMP* and a membership test, which is more expensive to compute than the assertion before this step. It turns out that this seemingly expensive result can be transformed further into one with aggregation function *max* and derived attribute *MaxSal*. Hence it provides opportunities for great efficiency improvement, with advanced optimization techniques such as finite differencing[14].

Finally, although RAINBOW is able to generate all possible valid reformulations of a given constraint, it does not tell us which one is the best in terms of computational efficiency. Nor does it tell us how to find such a reformulation efficiently. In the next two sections, we present a cost function, which serves as a criterion for selecting antecedents, and our strategies for search control. Both control knowledge and efficiency knowledge are represented as rules which are stored in the knowledge base, together with the domain-specific knowledge used to derive antecedents. Such a representation of meta-knowledge makes the system self-extensible and offers great ease in adapting to new environments and incorporating new knowledge[7,20]. Sample rules can be found in [15].

4. Cost Analysis and Measurement

In searching through the space of possible constraint reformulations, a criterion is needed for choosing the best solution (antecedent), predicting the best search direction, and pruning unpromising branches. In this section we investigate the factors on which such a criterion depends and propose a method of measurement that takes into account all the relevant information. Basically we prefer one reformulation over another according to its computational cost and semantic weakness. We discuss them separately using the techniques of symbolic and incremental analysis. The design goals for such a cost function are to make the process fully automatic, minimizing the need for the user to provide performance information. The accuracy of the measurement relies heavily on the accuracy of information provided by the user or through monitoring the database in operation, and the correctness of assumptions made, such as the independence of user-defined predicates.

Computational Cost

We define the computational cost of a constraint to be the time to check its validity against a database state. Four pieces of information are used as parameters to our cost formula. We need the estimated selectivities of user-defined predicates, monitored sizes of relations and attribute images, and monitored frequency information about relation update operations — insertion and deletion frequencies of single tuples. These parameters are initially specified via one of two means: (1) The user may provide such information, or (2) the system may assume default values for those that are missing. Subsequently the system computes these parameters for other predicates or relations that are defined in terms of the initial set of predicates and relations, and continuously modifies them to reflect the state change. Such activity is specified by sets of transformation rules[15].

The cost of computing an integrity constraint has two components: (1) the cost (c_1) of actually evaluating the constraint weighted with the frequency that the domains (relations) it constrains on change, and (2) the cost (c_2) of computing and maintaining the auxiliary structures used in the constraint, e.g., access paths and derived information, weighted with the frequencies that the domains those structures depend on change. The general cost formula for computing the cost of constraint P is:

$$c(P) = c_1(P) + c_2(P), \quad \text{where}$$
$$c_1(P) = c_{eval}(P) \times \sum_{x \in DOM(P)} (f_{ins}(x) + f_{del}(x))$$
$$c_2(P) = \sum_{y \in STR(P)} (c_{main}(y) + c_1(def(y)))$$

In the above formula, $DOM(P)$ is the set of relations in terms of which the constraint P is specified, $STR(P)$ is the set of auxiliary structures used in P, $f_{ins}(x)$ and $f_{del}(x)$ are the insertion and deletion frequencies to relation x, $c_{eval}(P)$ is the cost of evaluating the expression P, $c_{main}(y)$ is the cost of maintaining structure y, and $def(y)$ is the definition formula of y. This formula achieves a good compromise between performance of enforcing constraint and performance of maintaining redundant information. There are two groups of transformation rules[15]: (1) Rules for computing the cost of evaluating arbitrary expressions, and (2) Rules for estimating the cost of maintaining arbitrary materialized structures (e.g., views).

Semantic Weakness

As mentioned in Section 2, the reformulation process has to be retracted when some knowledge used in reformulating constraint P into P' (1) has become invalid, or (2) is an implication instead of an equivalence, and P' is false, although the database after the state transition is still valid in terms of the original constraint P. The assertion h5 in Figure 1 is such an example. Both situations are because that we are using some facts which are semantically stronger than necessary, some facts which only hold for a subset of valid database states. We need some means to measure the semantic weakness of each piece of knowledge in the knowledge base and determine, according to such a measure, whether it is cost-effective to use a particular one in our reformulation.

We take the semantic weakness of an assertion to be the probability that it is true in a valid database state. The larger this probability is, the weaker the assertion is in semantics. In reformulating a constraint, we want to use a piece of knowledge which is as weak as possible such that our reformulation has less chance of having to be retracted. In establishing the knowledge base, each assertion is attached with the probabilty that

it will remain true as its initial semantic weakness measure. For each assertion which is an implication $P \Rightarrow Q$, its weakness measure is multiplied by the probability that $Q \Rightarrow P$ is true as its actual measure. This is because if we use $P \Rightarrow Q$ in reformulation we are assuming that $Q \Rightarrow P$ is also true. These initial probability measures are specified by the user at knowledge base build time. Transformation rules are applied to compute the semantic weakness of derived knowledge and partially reformulated constraints.

Derived Assertions:

d6. $EName(e) = AEmp(a)$
d7. $AProj(a) = PName(p)$
d8. $PDept(p) = DName(d)$
d9. $e = IEName(AEmp(a))$
d10. $p = IPName(AProj(a))$
d11. $d = IDName(PDept(p))$

Goal:

G2. $Sal(e) \leq Bgt(d); H \cup \{d1\}$
↓ R3+d1
G11. $Sal(e) \leq Bgt(d); H \cup \{d1, d6, ..., d8\}$
↓ R7+h6+d6, R7+h7+d7, R7+h8+d8
G12. $Sal(e) \leq Bgt(d); H \cup \{d1, d6, ..., d11\}$
↓ R9+d9, R9+d11
G13. $Sal(IEName(AEmp(a))) \leq Bgt(IDName(PDept(p))); H \cup \{d1, d6, ..., d11\}$
↓ R9+d10
G14. $Sal(IEName(AEmp(a))) \leq Bgt(IDName(PDept(IPName(AProj(a))))); H \cup \{d1, d6, ..., d11\}$
↓ P1

Figure 4: An $\{a\}$-*antecedent* of the goal

Given an assertion P and its semantic weakness measure $w(P)$, the modified cost formula which takes into account the semantic weakness of the assertion is as follows:

$$c'(P) = a \times c(P)/w(P)$$

where a is a predetermined constant which balances the compromise of semantic weakness against other cost factors. For example, we may associate with the assertion h5 in Figure 1 a semantic weakness measure $p \times q < 1$ where (1) the probability of h5 remaining true is p and (2) the probability of

$$(EDept(e) = DName(d)) \Rightarrow (\exists p)(\exists a)[PDept(p) = DName(d) \wedge EName(e) = AEmp(a) \wedge AProj(a) = PName(p)]$$

being true is q. w increases the cost of the reformulations in Figures 2 and 3 and at certain point they are no longer cheaper than some other reformulations which do not use h5. Figure 4 shows such a derivation.

5. Search Control

The search space for reformulating a constraint is the space of constraints which are reformulations of the original constraint. The stepwise-reduction approach associates a natural search tree with each constraint's search space. Most of the decisions made during the search are based on the measurement of computational cost and semantic weakness. This suggests the adoption of the basic paradigm of heuristic search. The root node of the tree is the initial constraint to be reformulated. The arcs represent the application of reduction rules. The branching points in the tree represent points where more than one reduction rule is applicable. Intermediate nodes are partial reformulations and leaves are alternative constraints to be enforced. Figure 5 shows a partial search tree for our example constraint, where the branches correspond to the three derivations in Figures 2, 3, and 4.

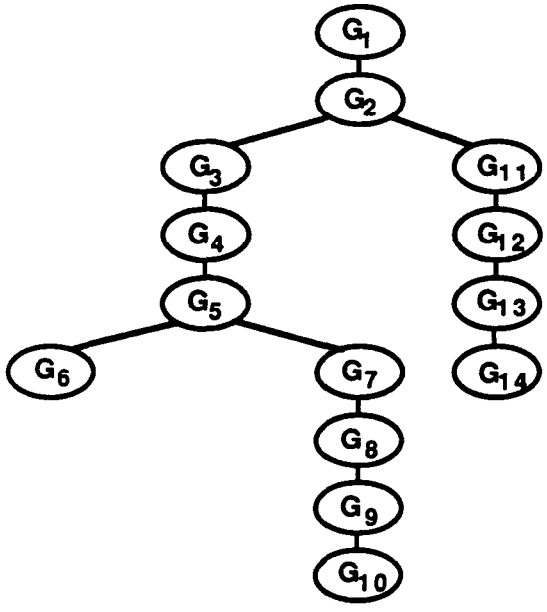

Figure 5: Partial Search Tree

The basic search technique is a form of heuristic search with the state of the search recorded in a task agenda. A task is defined to be an application of a rule to a goal. A reformulation node is chosen based on a modified form of best-first search. The search attention is always concentrated on the current node unless it is abandoned according to the pruning criteria. Task-ordering rules are used to choose a task within the current node to work on. Only those tasks which are considered plausible are taken as candidates. A reduction rule is then applied to the current node to fulfill the chosen task. The resulting new node is compared to other possibilities in the tree by a form of branch and

bound.

Plausible Task Generation

A critical part of constraint reformulation is the decision of what to do when more than one reduction rule is applicable. Each rule usually represents the possibility of a different way of reformulating the constraint. Two types of action are possible at this point. All the applicable rules can be applied, producing the set of competing reformulations for comparison. Or, the rules can be compared without actually applying them, and those rules which are not meaningful for the purpose of reducing the cost of constraint validation is removed. Obviously the latter choice is more efficient. We use plausibility rules as a first-level filter to eliminate those candidate rules which do not lead to potential reduction in validation cost. Below we describe some of the rules in more detail.

Structures introduced should not be irrelevant to the original constraint. One of the important characteristics of constraint reformulation is the introduction of auxiliary structures or information into the constraint, which are maintained by the DBMS. This is accomplished through substitution of equal terms (R9) or forward inferencing (R7). But blindly introducing new structures may lead to more expensive constraints. One type of rules tries to avoid the introduction of irrelevant structures or domains. For example, if we apply reduction rules R5, R8(a), and R3 to the example constraint to get a derived assertion $AProj(a) = PName(p)$, and there is another assertion:

$$AProj(a) = PName(p) \Rightarrow (AEmp(a) = PMgr(p) \Rightarrow Percent(a) \geq 50)$$

then we do not want to infer $AEmp(a) = PMgr(p) \Rightarrow Percent(a) \geq 50$ because that it is not directly relevant to our goal $Sal(e) \leq Bgt(d)$.

Structures introduced should help in constraining the range of the original constraint. A constraint is always defined on a set of domains. The cost of checking the constraint is proportional to the product of the sizes of these domains. If a piece of knowledge specifies a restriction on these domains, combining it with the constraint will reduce the evaluation cost. As an example discouraged by such type of rules, suppose we have an assertion $EName(e) = SKILLOF(SEmp(s))$ which says that there is a mapping $SKILLOF$ from each $SKILL$ tuple to the EMP tuple that has that skill. If we replace $EName(e)$ in our example constraint by $SKILLOF(SEmp(s))$, we are introducing a new domain $SKILL$ which is not one of the domains of the original constraint.

General optimization. Most general-purpose, context-independent optimization techniques can also be specified as plausibility rules, such as finite differencing, pushing unary operations through binary ones, etc.

Task Ordering Mechanisms

There may very well be more than one plausible reduction task applicable to a reformulation node. The order in which decisions are considered would not be important if all combinations of possibilities were considered in full detail. However, the computation involved in choosing a reformulation must be limited. Therefore the ordering of tasks becomes relevant. The goal of task ordering rules is to reach the best solution as soon as possible. These rules try to compare the potential impact of tasks on the cost of constraint and order them accordingly.

Introduce stable structures first. The stability of structures are evaluated using the database monitoring information about the accessing and updating frequencies of domains on which the structures are defined. It also depends on the specific techniques used in implementing these structures. One ordering principle is to perform the task which introduces stable structures. By requiring the new structures to be stable, we are sure that the reformulation would have a low maintenance cost. The way of determining the stability of structures is to compare the cost of maintaining them. As an example of this ordering principle, consider the search tree in Figure 5. The choice point at node G5 indicates that two tasks are applicable. One introduces the auxiliary structure $EMPIN$ while another introduces $HASEMP$. If $HASEMP$ is implemented in a more expensive way than $EMPIN$, then according to our ordering criterion, G6 is prefered over G7.

Replace expressions that are expensive. Another ordering principle is to choose the task that replaces a more expensive expression. By replacing a more expensive expression first, we expect to reduce the cost of constraint faster. For example, at node G2 in Figure 5, two tasks are applicable. One task generates d2 from h5 and d1 while another generates three assertions d6, d7, and d8 from d1. Obviously h5 and d1 together are more expensive to evaluate than d1. Using our principle, the branch to node G3 is prefered.

Simplification should be done before looking at other choices. A reduction is said to be a simplification step if it symplifies the syntax of the constraint. One of the characteristics of constraint reformulation is that the benefit of reformulation may not be obvious by a single reformulation step. Some simplification steps are usually needed before there is a decrease in cost. By grouping one reformulation step with a sequence of simplification steps, we are making "macro step" reformulations which often help us find the right solution faster. Consider again the search tree in Figure 5. If, at node G8.

after $HASEMP$ has been introduced, we stop because the cost of the constraint at G8 is greater than the cost of the constraint at node G6, we would not be able to find a perhaps even cheaper constraint, G10.

Independent reductions should not be done repeatedly. Reformulation is a chain of reductions from one constraint to another. During this reduction process, there may be many independent decisions. These independent decisions should be made only once according to the order of their potential impact on the cost of evaluating constraint. For example, in the derivation in Figure 4, we did not show the ordering of derivations of d6, d7, and d8 from G2 to G11. The ordering is not important for reformulation purposes because they are independent of each other. RAINBOW automatically avoids the repeat derivations of these independent decisions in different orders.

Branch and Bound

The ordering of tasks still does not make sense if we have no means of determining when we have reached a "best" solution. Without a criterion for stopping the search, we again end up generating the whole search space exhaustively. The traditional technique of branch and bound does not work very well with reformulation in general because it requires the estimation of the upper and lower bounds of the cost of a subtree without actually generating the tree. Such cost bounds are very hard to obtain due to the nature of reformulation. At a particular node in the search tree, it is extremely difficult, if not impossible, to "guess" what is an alternative form into which the current constraint can be reformulated, just by looking at the constraint itself. Furthermore, the cost of the constraint is not uniformly decreasing during reformulation. Therefore it is of no use comparing the cost of nodes in the partially generated search tree.

Based on the above considerations, we have developed two strategies for branch pruning in the search process. One pruning principle makes use of the fact that a constraint specifies a relationship between objects in different domains, and according to our plausibility rules no new domain is ever introduced. Hence in the worst case the constraint is enforced exactly on these domains with no auxiliary structures to take advantage of, which means that the constraint has to be enforced by enumerating over all the domains involved. On the other hand, the best we may get is to enforce the constraint on a single smallest domain. At each node in the search tree these upper and lower bounds are computed by looking at the domains on which the current constraint is specified. If we denote the upper and lower cost bounds of a node N by c_{max} and c_{min} respectively, a node N is pruned if there exists another node N' such that $c_{max}(N') < c_{min}(N)$. For example, in Figure 5 $c_{min}(G13) = size(PROJ)$ while $c_{max}(G6) = size(EMP)$. If G6 is generated before G13 and $size(EMP) < size(PROJ)$ then G13 will be pruned and G14 would not be generated.

Another method is to have the user provide some criteria to stop the search. One such information would be a cost bound, where a solution is satisfactory if it costs less than the user's bound. A node gets pruned if its lower bound is larger than user's bound. Another criterion may be the set of domains that the user wants the constraint to be specified. For example, the user may tell the system that an $\{x, y\}$-*antecedent* is satisfactory. As soon as such a constraint is reached, the search stops.

6. Discussion

A prototype system is being implemented on top of RAINBOW using a wide-spectrum programming language $REFINE^{TM}$*. There are currently about 90 rules in the cost evaluator grouped according to functionalities and about 10 meta-rules in search controler. The heuristic rules for controling forward inferencing turn out to be hard to implement due to RAINBOW's difficulty in switching inference directions. We've tested the system using our example database and constraint, with a knowledge base of 20 assertions. RAINBOW is able to generate all alternative constraints, and based on given performance parameters the system correctly chooses the reformulated constraint in Figure 2 as the best solution. With the limited control facility we have, the search is roughly 10 times faster. The search space is non-trivial and deserves good search strategies. Although the reformulation of constraints as we stated here has a nice match to the general heuristic mechanism, the heuristics are of a quite different nature and rely heavily on the semantics of the application that the database models.

The problem of choosing between alternative reformulations is quite important, since it affects the efficiency of both the search process and the resulting constraint. In our framework, this problem has been broken into two components: (1) the logic component which constructs the search space whose nodes are reformulations of the original constraint, and (2) the control component which explores this space, making choices based on the cost of the alternatives. The first function is provided by RAINBOW's deductive reasoning paradigm, the second is provided by a combination of analytical and heuristic paradigms. These paradigms are combined in a uniform way to achieve the ultimate goal.

* A trademark of Reasoning Systems Inc.

We argue that the enforcement of integrity constraints based on the knowledge about application domain and database configuration is the right approach to the problem due to the nature of integrity constraint. We also demonstrated the feasibility of such an approach by formalizing and developing a framework for carrying out the reformulation effectively.

Acknowledgement

We are grateful to Gio Wiederhold, Cordell Green, Tom Pressburger, and Janet Coursey for helpful discussions. This work was supported in part by DARPA contract N39-84-C-0211 for Knowledge Based Management Systems, and in part by the Rome Air Development Center (RADC) contract F30602-86-C-0026. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of DARPA, RADC, or U.S. Government.

References

[1] Badal, D. and Popek, G., "Cost performance analysis of semantic integrity validation methods"; *Proc. ACM SIGMOD*, 1979, 109-115.

[2] Bernstein, P., Blaustein, B., and Clarke, E., "Fast maintenance of semantic integrity assertions using redundant aggregate data"; *Proc. 6th Int. Conf. VLDB*, 1980, 126-136.

[3] El-Masri, R. and Wiederhold, G., "Properties of Relationships and their Representation"; *Proc. of the 1979 NCC*, AFIPS vol.49, Aug. 1979, 319-326.

[4] Furtado, A., dos Santos, D., and de Castilho, J., "Dynamic modelling of a simple existence constraint"; *Inf. Syst.* 6, 1981, 73-80.

[5] Hammer, M. and Zdonik, S., "Knowledge-based Query Processing"; *Proc. 6th Int. Conf. VLDB*, 1980, 137-147.

[6] Hsu, A., Imielinski, T., "Integrity Checking for Multiple Updates;" *Proc. ACM SIGMOD Conf.*, 1985, 152-168.

[7] Kant, E., *Efficiency in Program Synthesis*; UMI Research Press, 1981.

[8] Keller, A.M. and Wiederhold, G., "Validation of Updates Against the Structural Database Model"; *Proc. of Symposium on Releability in Distributed Software and Database Systems*, Pittsburgh, July 1981.

[9] King, J., "Quist: a System for Semantic Query Optimization in Relational Databases"; *Proc. 7th Int. Conf. VLDB*, 1981, 510-517.

[10] Lafue, G., "Semantic Integrity Dependencies and Delayed Integrity Checking"; *Proc. 8th Int. Conf. VLDB*, Mexico City, 1982.

[11] Morgenstern, M., "The Role of Constraints in Databases, Expert Systems, and Knowledge Representation;" *Proc.1st workshop on Expert Database Systems*, Oct. 1984.

[12] Nicolas, J., "Logic for Improving Integrity Checking in Relational Data Bases"; *ACTA Informatica* 18, 227-253, 1982.

[13] Nicolas, J. and Yazdanian, K., "Integrity Checking in Deductive Databases"; *Logic and Databases*, eds.Gallaire, H. and Minker, J., Plenum Press, NY, 1978.

[14] Paige, R., "Applications of finite differencing to database integrity control and query / transaction optimization"; *Advances in database theory, Vol.2*, ed. H. Gallaire, J. Minker and J. Nicolas, Plenum Press, New York.

[15] Qian, X. and Smith, D., "Constraint Reformulation: An Approach to Efficient Validation"; *Tech. Report*, Stanford University and Kestrel Institute, 1987.

[16] Qian, X. and Wiederhold, G., "Knowledge-based Integrity Constraint Validation"; *Proc. 12th Int'l Conf. VLDB*, Kyoto, Japan, Aug.1986.

[17] Shepherd,A. and Kerschberg,L., "Constraint Management in Expert Database Systems"; in *Expert Database Systems*, ed. L. Kerschberg, Springer-Verlag, New York, 1984.

[18] Smith, D.R., "Derived Preconditions and Their Use in Program Synthesis"; ed. D. Loveland, *6th Conf. Automated Deduction*, Lecture Notes in Computer Science no.**138** (Springer-Verlag, New York, 1982), 172-193.

[19] Smith, D.R., "Top-Down Synthesis of Divide-and-Conquer Algorithms"; *Artificial Intelligence*, **27**(1) Sept.1985, 43-96.

[20] Smith, D.R., Kotik, G.B., and Westfold, S.J., "Research on Knowledge-based Software Environments at Kestrel Institute"; *IEEE Trans. on Software Engineering*, SE-**11**(11):1278-1295, Nov.1985.

[21] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification"; *Proc. of the 1975 SIGMOD Conference*, ACM SIGMOD, San Jose, June 1975.

[22] Wiederhold, G., "Knowledge versus Data"; in *On Knowledge Base Management Systems: Integrating AI and Database Technologies*, ed. M. Brodie, 1986.