1990

A NEW APPROACH FOR CONFLICT RESOLUTION AND RULE PROCESSING IN A KNOWLEDGE-BASED SYSTEM

Akhil Kumar
Cornell University

A NEW APPROACH FOR CONFLICT RESOLUTION AND RULE PROCESSING IN A KNOWLEDGE-BASED SYSTEM

Akhil Kumar
S. C. Johnson Graduate School of Management
Cornell University

ABSTRACT

In a knowledge-based system, rules can be defined to derive virtual attributes. Conflicts occur if multiple rules are applicable and one must be selected based on some criterion, such as priority. We identify important properties of a conflict resolution method and describe a technique for resolving conflicts and efficiently processing queries involving virtual attributes in a knowledge-based system. It is shown that by transforming a given, prioritized set of rules into a conflict-free, priority independent form it is possible to do query processing in a set-at-a-time manner. Algorithms for conflict resolution and query processing are given.

1. INTRODUCTION

One way to build a knowledge-based system is to extend a conventional database system by providing an ability to define, represent and process rules, in addition to data. These rules are in the nature of both integrity constraints that validate the integrity of the data in the database and derivation rules that allow new data to be derived. In such a system, processing a query consists of retrieving some attributes which are already stored in relations, and **virtual attributes** that are not explicitly stored, but derived by applying rules. The knowledge-based system must process rules efficiently, i.e., identify the appropriate rule and fire it to derive a virtual attribute.

An important issue in rule processing is conflict resolution. A conflict occurs when more than one rule is applicable in deriving the value of an attribute. Consider a complex rule such as: the salary of an employee is 1,000 times his age; however, if his name is Smith he earns 50K; besides, if he works in the sales department then he earns 60K and finally, if his name is John he earns the same salary as Mike. In a knowledge-based system, it is necessary to find an efficient way of representing such complex rules, perhaps as several simple rules, and then to resolve any conflicts correctly while processing a query to determine the value of a virtual attribute. Whenever multiple rules are applicable, the appropriate one must be selected based on some criterion or conflict resolution scheme. The criterion could consist of selecting the rule with the highest priority among the competing rules as in Postgres (Stonebraker, Hanson and Potamianos 1988) or it could be more complex as in OPS5 (Forgy 1981). (See Ioanaddis and Sellis [1989] for a discussion of alternative conflict resolution criteria.)

While considerable research effort has been devoted to integrating rules, expressed in a Prolog-like language, with a database system, and on optimizing recursive queries (Ullman [1989] gives an excellent coverage of this research), the topic of conflict resolution in databases has received little attention. In fact, the main focus of the techniques like the ones presented by Ullman (1985), Kellogg, O'Hare and Travis (1986), Tsur and Zaniolo (1986), and Morris, Ullman and Van Gelder (1986) is on developing methods for query processing by optimally compiling a set of rules.

The conflict resolution scheme is important because it has a direct impact on the query processing method which, in turn, could affect the degree of concurrency in a multi-user system. In this paper, we propose a method for resolving conflicts in a database system extended with an ability to define rules and show how it leads to efficient processing of non-recursive queries.

In the, OPS5 expert system, the Rete algorithm (Cooper and Wogrin 1988) is an important component of the inferencing and conflict resolution strategy. Even though, as illustrated by Sellis, Chen and Raschid (1988), some aspects of the Rete algorithm can be implemented more efficiently within a database system, there are fundamental differences between a database system and an expert system. An expert system is used for consultation purposes and usually a dialogue between a user and the system takes place. During the dialogue, the expert system elicits responses from the user and then applies various rules in order to reach conclusions. On the other hand, a database system is used primarily for processing queries, often ad hoc ones, on very large amounts of data stored in tables. During query processing, rules are applied to derive additional information not explicitly stored. Consequently, the rule processing strategy must be different in the two situations.

Three essential features of a rule-based system are set-at-a-time processing, order independence, and invertibility. **Set-at-a-time** processing means that the system can operate on a collection of tuples or an entire relation, rather than processing one tuple at a time. This is necessary for efficient processing of large amounts of data. The **order independence** property refers to the ability to process a group of rules in any order, independent of the priority associated with a rule. Finally, **invertibility** is the ease of identifying tuples that a given rule applies to. For instance, given a collection of rules to derive the department of an employee, it should be possible to easily find all employees who work in department 'd1'. Therefore, a good conflict resolution scheme should make it possible to process rules in this manner.

The organization of this paper is as follows. In Section 2, we present our model for defining and representing rules in a database system. Section 3 describes our conflict resolution scheme and gives an algorithm for converting a collection of rules into a suitable form. In Section 4, we illustrate how this scheme affects query processing performance. Finally, a conclusion and directions for future work are presented in Section 5.

2. REPRESENTING RULES

2.1 Basic Model and Assumptions

Several proposals for more powerful, next-generation database systems have emerged in the last few years, for example, Postgres (Stonebraker and Rowe 1986), HiPAC (McCarthy and Dayal 1989), Starburst (Lindsay, McPherson and Pirahesh 1987). A common aspect of most of these proposals is support for rules. Another feature is support for events, whereby both internal and external events can trigger certain actions to take place, thus making the database system "active." In this subsection, we shall review our basic model and state our assumptions.

We assume an event-condition-action (ECA) model similar to the one described by McCarthy and Dayal (1989) for the HiPAC database system. The approach described by Stonebraker, Hearst and Potamianos (1989) and Widom and Finkelstein (1989) is also similar. In this model, rules are triggered by events; i.e., upon the occurrence of a certain event E, the database checks if the condition C is true, and if so, action A is executed. For example:

Event:	insert employee tuple
condition:	if employee.salary > 100,000
action:	employee.rank = "top-management".

In this example, when a new tuple is inserted into the employee relation, the system checks if the salary of the new employee is greater than 100,000 and, if so, assigns the value "top-management" to the rank attribute. The event could also be external to the database as long as an event detector module can recognize it and send a message to the database system. The remainder of this paper faeces only on the condition and action parts of a rule.

We further assume that, in general, several rules are defined for deriving a virtual attribute and the condition and action parts of each rule are expressed in a language similar to SQL (Date and White 1989). More specifically, the condition clause C is an SQL predicate and the action part A is an SQL statement, as we illustrate by examples in the next subsection.

2.2 Representation Scheme

In this section, we discuss a scheme for representing rules in a database system. For example, consider the following set of three rules for deriving the department attribute of an employee from his name and salary attributes (the rules are in increasing order of priority from R1 to R3):

R1: $\rightarrow$ emp.dept = d1
R2: emp.name = "mike" $\rightarrow$ emp.dept = d2
R3: emp.salary > 60K $\rightarrow$ emp.dept = d3

This set of rules means that, given an employee tuple, rule R3 must be tried first, and then R2, in order to derive the department of the employee. If both R3 and R2 fail, then by default, R1 determines the department.

One method for representing these three rules, in four relations, is illustrated in Tables 1 through 4. The **Priority** relation contains the rule name and its priority. A higher number corresponds to a larger priority. The condition part of the rule is the WHERE predicate of SQL, and is written as a conjunction of terms, stored in the **condition** relation. The **action** relation contains a rule name, and the symbol of the action to be performed if the rule is activated or fired. Finally, the **Data** relation contains the full SQL expression for each condition and action symbol present in the condition and action relations.

Table 1. Priority Relation

Rule No.	*Priority*
R1	1
R2	2
R3	3

Table 2. Condition Relation

Rule No.	*Condition*
R2	c_1
R3	c_2

Table 3. Action Relation

Rule No.	*Action*
R1	a_1
R2	a_2
R3	a_3

Table 4. Action Relation

symbol	*SQL translation*
c_1	emp.name = "mike"
c_2	emp.salary > 60K
a_1	emp.dept = d1
a_2	emp.dept = d2
a_3	emp.dept = d3

The problem with this representation scheme is that it is not amenable to set-at-a-time processing. Consider a query to retrieve the name, salary, and department of all employees given the above set of rules. The system would examine one tuple at a time, first identifying all rules that apply to it. Then, one of the applicable rules is selected and fired, i.e., the specified action is performed to determine the value of the virtual attribute, in this case the department. Clearly, if the size of the relations involved is large, this tuple at a time approach is very slow.

2.3 Improved Representation

We propose to rewrite a given set of rules in a new form which is more efficient for query processing. Basically, we propose that a set of rules defined by a user for deriving a virtual attribute be transformed by the database and stored in a conflict-free form. For example, the three rules above, for deriving an employee's department, are rewritten as:

R1: emp.name ≠ "mike" and emp.salary ≤ 60K → emp.dept = d1

R2: emp.name = "mike" and emp.salary ≤ 60K → emp.dept = d2

R3: emp.salary > 60K → emp.dept = d3.

Tables 5 and 6 show the revised condition and data relations corresponding to the revised rules. The revised representation has two useful properties. First, in this form, only one rule would apply to any given tuple because $C_i \wedge C_j = \emptyset$ for all i,j, i.e., the condition part of any pair of rules is mutually exclusive. Secondly, the new set of rules is priority independent. Both of these properties aid in efficient query processing.

Table 5. Revised Condition Relation

Rule No.	*condition*
R1	c_3
R1	c_4
R2	c_1
R2	c_3
R3	c_2

Table 6. Revised Data Relation

symbol	*SQL translation*
c_1	emp.name = "mike"
c_2	emp.salary > 60K
c_3	emp.salary ≤ 60K
c_4	emp.name ≠ "mike"
a_1	emp.dept = d1
a_2	emp.dept = d2
a_3	emp.dept = d3

2.4 Desired Semantics

Given a collection of n rules $R1,...,Rn$, in increasing order of priority, we wish to rewrite each Ri as Ri'. The desired semantics from the new set of rules is stated in the following two conditions:

Condition 1:

$$C_i'(t_j) = \mathit{true} \text{ iff } C_i(t_j) = \mathit{true} \text{ and } C_k(t_j) = \mathit{false},$$
$$\mathit{for}\ i + 1 < k < n,\ a_k \neq a_i.$$

Condition 2:

$$(C_i'(t_j) \wedge C_k'(t_j) = \emptyset) \vee a_i = a_k$$

$C_i(t_j)$ is the boolean result of applying the condition part of rule Ri to t_j, a tuple of the relation to which the virtual attribute belongs. The first condition states that if $C_i(t_j)$ is true for more than one rule from the original set, then $C_i'(t_j)$ must be true for the highest priority rule among them. $C_i'(t_j)$ can be true for a lower priority rule only if it has an identical action part as the highest priority rule. The second condition follows from the first and means that if $C_i'(t_j)$ and $C_k'(t_j)$are true for two different rules i and k, then the action part of both the rules must be identical.

We shall presently show that it is possible to rewrite a given set of rules in such a revised form. To see how this approach facilitates efficient query processing, consider the example query: **Select * from emp.** In order to process this query, we only need to modify it by appending the condition parts of each revised rule (R1 through R3 of the previous section) independently as a predicate to it, and then run the three new queries. Query modification would generate the following queries:

```
Q1: insert into answer
        select emp.name, emp.salary, "d1" from emp
        where emp.name ≠ "mike" and emp.salary ≤ 60K

Q2: insert into answer
        select emp.name, emp.salary, "d2" from emp
        where emp.name = "mike" and emp.salary ≤ 60K

Q3: insert into answer
        select emp.name, emp.salary, "d3" from emp
        where emp.salary > 60K
```

The **answer** table is a temporary relation for storing the final result with the same attributes as the target list of the original query. Finally, notice that the queries Q1, Q2 and Q3 can be optimized by conventional methods and run in any order.

3. CONFLICT ELIMINATION

In Section 3.1 we first show that a collection of rules can always be rewritten in an order-independent, conflict-free form, while Section 3.2 describes a procedure for negating SQL predicates. An algorithm for transforming a given set of rules is discussed in Section 3.3 and illustrative examples are given in Section 3.4.

3.1 Basic Approach

We assume a rule is expressed as:

R: if C then A (or C → A)

where

C: condition part, or the WHERE predicate in SQL.
A: action part, a SQL statement.

In general, the predicate C is a conjunction of individual terms or sub-clauses c_i. Therefore, it is expressed as:

$$C = c_1 \wedge c_2 \ldots \wedge c_n$$

If a disjunction is present in C, it is removed by rewriting the original rule as two or more rules. For instance,

R: if $(c_1 \vee c_2) \wedge (c_3 \vee c_4)$ then A

is rewritten as:

R1:	if $c_1 \wedge c_3$ then A
R2:	if $c_1 \wedge c_4$ then A
R3:	if $c_2 \wedge c_3$ then A
R4:	if $c_2 \wedge c_4$ then A

Assume that a collection of n rules is written in the above form where the i^{th} rule Ri is a conjunction of n_i clauses, and is expressed as:

$$Ri : C_i = c_{i1} \wedge c_{i2} \ldots \wedge c_{ini}\ \mathit{then}\ a_i$$

Furthermore, without loss of generality, assume that rule R1 has the lowest priority and Rn the highest. The significance of priority is that if multiple rules are applicable in order to derive a certain attribute value, the one with the highest priority must be selected. A new set of rules can be constructed from the original set by rewriting Ri as follows:

$$Ri' : \mathit{if}\ C_i \wedge \overset{k=n}{\underset{a_k \neq a_i}{k=i+1}}\ \mathit{not}(C_k)\ \mathit{then}\ a_i$$

Rewriting the rules in this manner satisfies the two conditions of Section 2.4. In the new form, either the condition part of exactly one rule will be satisfied, or if multiple condition parts are satisfied, then the corresponding action will be the same in all those cases. We now show how not(C), the negation for a clause C, is computed.

3.2 Negations of SQL clauses

If C is given in the form: $C = c_i \wedge c_2, ..., c_n$, the computation of **not**(C) depends upon the syntax and structure of C. Four cases were identified and are discussed separately below. In all cases it is assumed that the rule is used to derive an attribute T.D, where T is the relation name and D is the name of the derived attribute.

Case 1: Non-quantified one-variable clause

In the first case, each c_i contains a reference to only one relation name. Therefore, c_i is of the form: *LHS_i op constant*. Not(C) is expressed as follows:

$$\begin{aligned} \text{not}(C) &= \text{not}\ (c_1 \wedge c_2 \ldots c_n) \\ &= \text{not}(c_1) \vee \text{not}(c_2) \ldots \vee \text{not}(c_n). \end{aligned}$$

where

$$\text{not}(c_i) = LHS_i \text{ not-op constant.}$$

Not-op is the negation for the operator **op**. Table 7 shows the negation of some standard operators.

Table 7. Negation of Operators

operator	*not-operator*
$=$	$\neq$
$<$	$\geq$
$\leq$	$>$
$>$	$\leq$
$\geq$	$<$
IN	NOT IN

Case 2: Non-quantified multi-variable clause

If one or more c_i contain multiple relation names, then not(C) is computed differently from above. Then,

not(C) = not exists (select * from T1, T2, ..., Tn where C).

where T1, T2, ..., Tn are names of the other relations (other than T) that appear in C.

Case 3: Existential Quantification

If c_i is of the form exists(...), then the negation is not exists(...). On the other hand, if c_i is of the form not exists(...), then the negation is exists(...).

Case 4: Universal Quantification

Queries involving the "forall" universal quantifier can be reexpressed in terms of the existential quantifier. See Date and White (1989) for details of how to perform this transformation. Therefore, this case is treated like case 3.

3.3 Algorithm

This algorithm converts a collection of rules, *R*1, ..., *Rn*, for deriving a virtual attribute into a non-conflicting form by the technique described above. The algorithm is run every time a change is made to the original set of rules.

As before, it is assumed that the rules are initially ordered by increasing priority from *R*1 to *Rn*. They are examined in descending order of priority, and the negation of each rule is computed and appended to all rules with lower priority than itself. The main steps in the algorithm are listed below.

```
k=n
while (k>1){
    compute not($C_k$) by the procedure of Section 3.2
    for (i = 1; i $\leq k - 1$; i++)
        if ($a_i \neq a_k$)
            append not($C_n$) to $C_i$;
    apply simplification rules
    k=k-1;
        }
```

The simplification step is meant to examine the possibility of rewriting the clause in a simpler form. If any of the simplification rules are applicable, then the clause can be rewritten in a simpler form. These rules are:

1. If $c_i \rightarrow c_j$, then $c_i \wedge c_j \rightarrow c_i$

   for example,

   $X = a \wedge X \neq b \wedge a \neq b \rightarrow X = a$

   X op a and X not-op a $\rightarrow$ unsatisfiable condition

2. If $c_i \rightarrow c_k$ then $(c_i \text{ or } c_j) \wedge (c_i \text{ or } c_k) \rightarrow c_i \text{ or } c_j$

   where c_i, c_j, and c_k are sub-clauses.

3.4 Examples

In this section we discuss two examples that illustrate our procedure for conflict elimination.

Example 1: Consider an employee relation **emp**, and a department relation **dept** as follows:

emp(Name, DNo., salary, group)
dept(DNo., Sales)

Assume that the group of an employee is a virtual attribute derived from his salary and his department's sales by the following rules (below, rule R4 has the highest priority and R1 the least):

R4: emp.DNo = dept.DNo and emp.salary > .03 × dept.sales → emp.group = "g1"

R3: emp.DNo = dept.DNo and emp.salary > .02 × dept.sales → emp.group = "g2"

R2: emp.DNo = dept.DNo and emp.salary > .01 × dept.sales → emp.group = "g3"

R1: → emp.group = "g4"

After applying our algorithm and the simplification rules of Section 3.3, this set of rules is rewritten as:

R4: emp.DNo = dept.DNo and emp.salary > .03 × dept.sales → emp.group = "g1"

R3: emp.DNo = dept.DNo and emp.salary > .02 × dept.sales and emp.salary ≤ 0.03 × dept.sales→ emp.group = "g2"

R2: emp.DNo = dept.DNo and emp.salary > .01 × dept.sales and emp.sales ≤ 0.02 × dept.sales → emp.group = "g3"

R1: not exists (select * from dept where emp.DNo = dept.DNo and emp.salary > .01 × dept.sales) → emp.group = "g4"

Notice that this new set of rules is priority independent.

Example 2: Consider another example with the following two relations:

emp(Name, DNo., salary)
dept(DNo, Sales)

In this example, assume that salary is a virtual attribute derived from the following rules (R2 has higher priority):

R2: exists (select * from dept where emp.DNo = dept.DNo) → emp.salary = .01 × sales from emp, dept where emp.DNo = dept.DNo.

R1: → emp.salary = 50K

These rules mean that an employee's salary is 1 percent of his department's sales if a department tuple exists for his department; else, it is 50K. This set of rules is rewritten in the following conflict-free form:

R2: exists (select * from dept where emp.DNo = dept.DNo) → emp.salary = .01× sales from emp, dept where emp.DNo=dept.DNo.

R1: not exists (select * from dept where emp.DNo = dept.DNo) → emp.salary = 50K

Again, the new set of rules is priority independent. Now we turn to discuss our query processing algorithm.

4. QUERY PROCESSING ALGORITHMS

In this section we discuss algorithms for processing queries involving virtual attributes, assuming the rules have already been transformed into a conflict-free form. Our strategy is based on query modification (Stonebraker 1975). Stonebraker (1975) has shown how query modification can be used to implement views and integrity constraints. Here the application of this technique to two types of queries involving virtual attributes is discussed: queries where virtual attributes are present only in the target list, not in the query predicate, and queries where virtual attributes are also allowed in the predicate. The first case is discussed in Section 4.1 and the second in Section 4.2.

4.1 Predicates without Virtual Attributes

In these queries, virtual attributes are allowed only in the target list of the query. Such a query is processed in two steps. The first step consists of splitting the relation whose attribute must be derived into fragments that satisfy each rule. This is performed by applying the condition part of each rule to the entire relation. The second step consists of actually computing the derived attribute by applying the action part of each rule to the appropriate fragment. If the action part assigns only a constant value to the virtual attribute, then the two steps can be combined into a single query. Otherwise, two queries are run for each rule. Consider a general query of the following form:

select target-list, T.D where Q

where

target-list: a target list of stored attributes
T.D: a virtual (or derived) attribute of relation T
Q: any qualification involving only stored attributes

For simplicity, assume that there is only one virtual attribute in the target list. The extension to the case of multiple virtual attributes is straightforward if they are independent of one another, i.e., a virtual attribute is not derived from another virtual attribute.

To run this query, the following steps are repeated as many times as the number of rules on the virtual attribute T.D. First, in **step 1**, relation T is fragmented horizontally into the tuples that satisfy rule *Ri'* and the fragment is stored in a temporary relation $Temp_i$ by running the following query:

```
insert into Temp_i
select * from T
where C'_i and Q.
```

Then, in **step 2**, the action part of rule *Ri'* is modified to create and run the following query:

```
insert into answer
select target-list, a_i($T=Temp_i).
```

Here a_i($T-$Temp_i$) refers to the action part of rule *Ri'* with the relation name T replaced by its fragment $Temp_i$.

The above two steps can be combined into one if the action part of a rule Ri assigns a constant value to the virtual attribute as in Example 3 below. In this case, the following query is generated:

```
insert into answer
select target-list, a_i from T, T1, T2, ..., Tm
where C'_i and Q.
```

T1, T2, ..., Tm are other relation names, in addition to T, that might appear in the predicate or the target-list of the modified query. The following two examples illustrate how this algorithm works.

Example 3:

Consider again the employee and department relations of Example 1. To answer a query **select * from emp**, we must apply the above algorithm independently to the four rules, R1 through R4 of Example 1. Since the action part of each rule assigns a constant to the group attribute, each rule generates only one query. The complete set of queries is as follows (R4 generates the query Q4, R3 generates Q3, and so on):

```
Q4: insert into answer
        select name, DNo,salary, "g1" from emp, dept
        where emp.DNo = dept.DNo and emp.salary >
            .03 × dept.sales.

Q3: insert into answer
        select name, DNo,salary, "g2" from emp, dept
        where emp.DNo = dept.DNo and emp.salary >
                .02 × dept.sales
            and emp.salary ≤ .03 × dept.sales.

Q2: insert into answer
        select name, DNo,salary, "g3" from emp, dept
        where emp.DNo = dept.DNo and emp.salary >
                .01 × dept.sales
            and emp.salary ≤ .02 × dept.sales.

Q1: insert into answer
        select name, DNo,salary, "g4" from emp
        where not exists (select * from dept where
                emp.DNo = dept.DNo
            and emp.salary > .01 × dept.sales)
```

The next example is different because the action part of one rule contains a complete SQL statement.

Example 4:

Consider the employee and department relations of Example 2. A query such as **select * from emp where emp.name = "mike"**, will be transformed into the following set of queries by our query processing algorithm (R1 generates Q1, while R2 generates two queries, Q21 and Q22):

```
Q1: insert into answer
        select name, DNo, 50K from emp
        where not exists (select * from dept where
                emp.DNo = dept.DNo)
            and emp.name = "mike"

Q21:    insert into Templ
        select name, DNo. from emp
        where exists (select * from dept where emp.DNo
                = dept.DNo)

Q22:    insert into answer
        select name, DNo., .01*sales from Templ, dept
        where Templ.DNo = dept.DNo and Templ.name
            = "mike".
```

4.2 Predicates Containing Virtual Attributes

The case where the predicate of a query contains a derived attribute is more complex. Although developing an efficient algorithm for processing such queries is a subject of our current research, we shall illustrate, with an example, how a special sub-class of such queries is processed.

Example 5: Consider an employee relation:

emp(name, salary, dept)

Assume that the virtual attribute department is derived by the following rules:

R1: $\rightarrow$ emp.dept = d1
R2: emp.salary > 60K $\rightarrow$ emp.dept = d2.

The conflict free representation of these rules is:

R1: emp.salary $\leq$ 60K $\rightarrow$ emp.dept = d1
R2: emp.salary > 60K $\rightarrow$ emp.dept = d2.

Now consider an SQL command to give a 10 percent raise to all employees in department "d1" expressed as:

```
update emp
set emp.salary = 1.1 × emp.salary
where emp.dept="d1".
```

One naive approach to process this command is to materialize the derived attribute for the entire relation and then process the query in a conventional manner. However, this is not efficient. Another approach is to append the predicate term that contains the derived field to the action part of each rule and select those rules where the result is true or non-null. Then the predicate in the user query is substituted by the condition part of each selected rule and the query is run. Consequently, in our example, rule R1 is selected as the relevant rule and the query is rewritten as:

```
Q1:     update emp
        set emp.salary = 1.1 × emp.salary
        where emp.salary ≤ 60K
```

This special case is easier to process because the action part of both rules consists of a constant assignment. The general case where any SQL statement can appear in the action is currently being investigated.

4.3 Discussion

In this section, we review the advantages of our approach. The main advantage in terms of query processing arises from the ability to do set-at-a-time processing. This is possible because the rules are written in a conflict-free form, leading to an efficient and simple algorithm for query processing.

Another advantage is that of greater concurrency. Consider Example 5 again and imagine that a second transaction to give a 15 percent raise to all employees in department d2 is run concurrently with the first. By the method described above, this transaction is transformed into the following query:

```
Q2:     update emp
        set emp.salary = 1.15 × emp.salary
        where emp.salary > 60K.
```

If there is an index on the salary field, then Q1 and Q2 can run concurrently because they will access non-overlapping tuples of the employee table. On the other hand, if rules were not transformed into a conflict-free form, then each transaction would have to access the entire table in order to determine which tuples are relevant. Therefore, not only will each transaction run slowly, but the two transactions will have to run in a serial order.

5. CONCLUSIONS

In recent years there has been considerable research in the design of more powerful database systems. One important feature of such "next generation" systems is the support for rules for deriving virtual attributes from stored data, rather than storing all attributes explicitly.

Such systems provide a facility for the user to define rules. A conflict occurs when multiple rules are applicable for deriving the value of a virtual attribute and, hence, one must be selected from them. This selection is made based on rule priority or some other criterion.

We have presented an approach for conflict resolution that involves transforming a set of rules into a new set in which conflicts are eliminated. It was shown that such a transformation is possible and an algorithm for doing so was described and illustrated. We have also shown that writing the rules in this manner makes it possible to process queries efficiently by query modification. Algorithms for processing queries involving virtual attributes were described and examples given. The ability to operate on sets of data instead of one tuple at a time leads to improved performance.

The algorithms for query processing presented here can be improved still further. More work is anticipated in developing query processing algorithms to handle the cases where the predicate of a user query contains a virtual attribute or where recursive queries are present.

We expect to design new indexing techniques and also draw upon some ideas from the area of semantic query processing in this effort.

6. REFERENCES

Cooper, T., and Wogrin, N. *Rule-Based Programming with OPS5.* San Mateo, California: Morgan Kaufman Publishers, 1988.

Date, C. J., and White, C. *A Guide to SQL/DS.* Reading, Massachusetts: Addison Wesley, 1989.

Forgy, C. *OPS5 Users' Manual.* Pittsburgh: Carnegie Mellon University Technical Report, 1981.

Ioanaddis, Y., and Sellis, T. "Conflict Resolution of Rules Assigning Values to Virtual Attributes." *Proceedings of the ACM SIGMOD Conference*, June 1989, Portland, Oregon, pp. 205-214.

Kellogg, C.; O'Hare, A.; and Travis, L. "Optimizing the Rule-Data Interface in KMS." *Proceedings of the Twelfth VLDB Conference*, September 1986, Tokyo, pp. 42-51.

Lindsay, B.; McPherson, J.; and Pirahesh, H. "A Data Management Extension Architecture." *Proceedings of the ACM SIGMOD Conference*, May 1987, San Francisco, pp. 220-226.

McCarthy, D., and Dayal, U. "The Architecture of an Active Database Management System." *Proceedings of the ACM SIGMOD Conference*, June 1989, Portland, Oregon, pp. 215-224.

Morris, K.; Ullman, J. D.; and Van Gelder, A. "Design Overview of the NAIL! System." *Proceedings of the Third International Conference on Logic Programming*, 1986.

Sellis, T.; Lin, C.; and Raschid, L. "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms." *Proceedings of the ACM SIGMOD Conference*, June 1988, Chicago, pp. 404-412.

Stonebraker, M. "Implementation of Integrity Constraints and Views by Query Modification." *Proceedings of the ACM SIGMOD Conference*, 1975, pp. 65-78.

Stonebraker, M., and Rowe, L. "The Design of Postgres." *Proceedings of the ACM SIGMOD Conference*, May 1986, Washington, D.C., pp. 340-355.

Stonebraker, M.; Hanson, E.; and Potamianos, S. "The Postgres Rules System." *IEEE Transactions on Software Engineering*, July 1988.

Stonebraker, M.; Hearst, M.; and Potamianos, S. "A Commentary on the Postgres Rules System." *ACM SIGMOD Record*, September 1989, pp. 12-19.

Tsur, S., and Zaniolo, C. "LDL: A Logic-based Data Language." *Proceedings of the VLDB Conference*, September 1986, Tokyo.

Ullman, J. D. *Principles of Database and Knowledge-base Systems, Volume II.* Potomac, Maryland: Computer Science Press, 1989.

Ullman, J. D. "Implementation of Logical Query Languages for Database Systems." *ACM TODS*, Volume 10, Number 3, September 1985, pp. 289-321.

Widom, J., and Finkelstein, S. "Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems." IBM Research Report, June 1989.