An Algebraic Approach to Rule Analysis in Expert Database Systems*

Elena Baralis†
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

{baralis,widom}@cs.stanford.edu

Abstract

Expert database systems extend the functionality of conventional database systems by providing a facility for creating and automatically executing Condition-Action rules. While Condition-Action rules in database systems are very powerful, they also can be very difficult to program, due to the unstructured and unpredictable nature of rule processing. We provide methods for static analysis of Condition-Action rules; our methods determine whether a given rule set is guaranteed to terminate, and whether rule execution is confluent (has a guaranteed unique final state). Our methods are based on previous methods for analyzing rules in active database systems. We improve considerably on the previous methods by providing analysis criteria that are much less conservative: our methods often determine that a rule set will terminate or is confluent when previous methods could not. Our improved analysis is based on a "propagation" algorithm, which uses a formal approach based on an extended relational algebra to accurately determine when the action of one rule can affect the condition of another. Our algebraic approach yields methods that are applicable to a broad class of expert database rule languages.

1 Introduction

In the past decade there has been a surge of interest in adding rule processing to database systems. *Deductive database systems* use logic rules to provide an expressive query facility [CGT90,Ull89]. *Active database systems* use Event-Condition-Action rules to provide reactive behavior [HW93]. In this paper we focus on what we refer to as *expert database systems*. An expert database system is a conventional database system extended with a facility for creating and automatically executing Condition-Action rules. Expert database systems originated by coupling a rule processor for a production rule language such as *OPS5* [BFKM85] to a conventional DBMS; this approach is taken in, e.g., [Tzv88]. More recently the prevalent approach has been to build rule processing directly into the database system. Examples of recent or ongoing projects in expert database systems are [BM93,DE89,DOS+92,GP91,SLR88]. Note that some systems described as active database systems actually use the Condition-Action rule paradigm, and hence fall into the class of expert database

*This work was partially performed while the authors were at the IBM Almaden Research Center, San Jose, CA. At Stanford this work was supported by equipment grants from Digital Equipment Corporation and IBM Corporation.

†Permanent address: Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24-10129 Torino, Italy.

systems as we use the term here; examples of such systems are [Han92, SKdM92]. Since expert database systems evolved from production rule systems such as OPS5 and are closely related to active and deductive database systems, the techniques presented in this paper certainly can be adapted for other database rule paradigms.

While expert database systems are very powerful, developing even small applications can be a difficult task, due to the unstructured and unpredictable nature of rule processing. During rule processing, rules can activate and deactivate each other, and the intermediate and final states of the database can depend on which rules are activated and executed in which order. It is highly beneficial if the rule programmer can predict in advance some aspects of rule behavior. This can be achieved by providing a facility that statically analyzes a set of rules, before installing the rules in the database [AWH92]. Static rule analysis can form the basis of a design methodology and programming environment for expert database systems.

As has been observed in the past [AWH92,KU94,vdVS93], two important and desirable properties of rule behavior are *termination* and *confluence*. A rule set is guaranteed to terminate if, for any database state and set of modifications, rule processing cannot continue forever (i.e. rules cannot activate each other indefinitely). A rule set is confluent if, for any database state and set of modifications, the final database state after rule processing is independent of the order in which activated rules are executed.

In this paper we propose a generally applicable algorithm for determining when the action of one rule can affect the condition of another rule. The algorithm uses an extension of relational algebra to model rule conditions and actions. Essentially, the algorithm "propagates" one rule's action through another rule's condition to determine how the action may affect the condition; hence, we call it the *Propagation Algorithm*. The Propagation Algorithm is useful for analyzing termination since it can determine when one rule may activate another rule. The Propagation Algorithm also is useful for analyzing confluence since it can determine when the execution order of two rules is significant. The Propagation Algorithm determines these properties much more accurately than previous methods, e.g. [AWH92,HH91,ZH90]. In addition, since we take a general approach based on relational algebra, our method is applicable to most expert database systems that use the relational model.

1.1 Previous Related Work

In traditional expert systems, i.e. production rule systems such as OPS5 [BFKM85], predicting properties such as termination and confluence is of far less importance than in the database environment; consequently, there has been little (or no) work on rule analysis in traditional expert systems. There are several reasons why predicting behavior is less important in traditional expert systems: Typically, they are stand-alone main-memory systems, so there is little fear of consuming resources or blocking other users. Since traditional expert systems are used for "intelligent inferencing"—deriving facts—the order of inferences or a unique final outcome may not be important. Finally, rule processing in traditional expert systems might last for hours and indeed may not terminate, but this behavior may be considered acceptable, unlike in the database environment.

In the database context, [HH91,ZH90] give methods for analyzing Condition-Action rules that are similar to the rules we consider. However, the goal of their work is to impose restrictions on rule sets so that confluence (a "unique fixed point" in their model) is guaranteed; we instead provide techniques for analyzing the behavior of arbitrary rule sets. In addition, the methods in [HH91, ZH90] have been shown to be weaker than the methods in [AWH92], which in turn are weaker than the methods we present here. The methods in [AWH92] are developed in the context of the Starburst Rule System, which uses an Event-Condition-Action (active database) rule model. Their technique for analyzing rule interaction relies on a shallow comparison of the actions performed by one rule and the events and conditions of another rule. We improve on this approach significantly by using a formal algebraic model that allows us to accurately analyze the interaction between rules using the semantics of rule conditions and actions. In an initial report we applied our approach to termination only [BCW93]; here we refine the techniques in [BCW93] and propose a general framework for analysis of both termination and confluence.

In other related work, [vdVS93] analyzes rule behavior in the context of object-oriented active database systems. Their work focuses on differences between *instance-oriented* and *set-oriented* rules (we consider only set-oriented rules in this paper) and on decidability properties for rule analysis. Their rule model is rather restricted, in that rule actions (methods) can only modify data selected by the corresponding rule condition, and deletions and insertions seem to be disallowed. The properties of confluence and of termination within some fixed number of steps are shown to be decidable using an approach based on "typical databases"; a typical database contains all possible data instances that could affect the outcome of rule processing. The rule set is "run" over the typical database and the outcome is checked for the desired properties. This approach is clearly infeasible in practical applications, so lower complexity algorithms are proposed, but the details and applicability of these algorithms are not clarified.

A rather different approach to rule analysis is taken in a recent paper [KU94], where Event-Condition-Action rules are reduced to *term rewriting systems*, and known analysis techniques for termination and confluence of term rewriting systems are applied. The rule model they use is quite different from ours, and it is unclear whether a general relational rule model such as ours can be expressed as a term rewriting system. However, in the future we plan to explore the relationship between these different approaches.

Our Propagation Algorithm is closely related to the problem of *independence of queries and updates*, addressed in, e.g., [Elk90,LS93]. [LS93], which subsumes [Elk90], gives an algorithm for detecting if the outcome of a query, expressed as a Datalog program, can be affected by a given insertion or deletion. For analyzing expert database rules, we need a somewhat stronger technique: when a query and update are not independent, we need to know whether the update adds to, removes from, or modifies the result of the query. Furthermore, while the algorithm presented in [LS93] applies to more general queries than we consider here (e.g. recursive queries), their model for database updates is considerably simpler than ours.

Finally, our Propagation Algorithm is somewhat related to *incremental evaluation*, as in [BW93, QW91,RCBB89]: both problems address the effect of a database modification on a relational expression. However, incremental evaluation techniques are designed for run-time, when the actual modifications are known, while our techniques apply at compile-time, when the modifications are expressed as database operations.

1.2 Outline of the Paper

In Section 2 we present our algebraic Condition-Action rule language and provide several examples that are used throughout the paper. Section 3 contains the Propagation Algorithm, examples of its application, and a proof of its correctness. In Sections 4 and 5 we apply the algorithm to the analysis of termination and confluence, respectively; again, several examples are included. In Section 6 we draw conclusions and outline future work.

2 Algebraic Rule Language

A rule in our language has a *condition* and an *action*. Rule conditions are expressed as queries over the database; rule actions are database modifications. We use a language in which conditions and actions are both represented by relational algebra expressions. In this section we describe the extensions to relational algebra that are required to represent general rule conditions and actions. Then we specify the syntax of our rule language using this algebra, and we describe the semantics of rule processing in our model. Finally, we give several examples of how Condition-Action rules may be represented in our algebraic language.

2.1 Algebraic Operators

Based on [CG85,Klu82], we define an extension to relational algebra that allows us to represent any queries that are expressible in SQL (or Quel), with the exception of the handling of duplicates and ordering conditions. We also introduce an extension that allows us to represent the SQL data modification operations **insert**, **delete**, and **update**.

Our extended relational algebra includes the basic relational algebra operators *select* (σ), *project* (π), *cross-product* ($\times$), *natural join* ($\bowtie$), *union* ($\cup$), and *difference* ($-$), which we do not elaborate on here; see [Ull89]. The first two lines of Table 1 present useful operators derived from the basic operators, while the next three lines present additional operators that we use. In the table, X and A denote attributes, B, A_1, and A_2 denote attribute lists, a is an aggregate function, and *expr* is an expression (explained below). In line 1, $E_1 \ltimes_p E_2 = \pi_{schema(E_1)}(\sigma_p(E_1 \times E_2))$; in line 3, α renames the attributes in list A_1 as A_2. In the remainder of the paper, we adopt the shorthand notation $E_1 \ltimes E_2$ and $E_1 \ltimes_{\nexists} E_2$ to denote $E_1 \ltimes_p E_2$ and $E_1 \ltimes_{\nexists p} E_2$ when predicate p equates all attributes in both $schema(E_1)$ and $schema(E_2)$ (similar to the natural join). We now discuss the other operators in more detail, then we present the modification operations.

Operator	Description
$\ltimes_p$	semijoin with predicate p
$\ltimes_{\nexists p}$	not-exists semijoin with predicate p
$\alpha_{A_1;A_2}$	attribute rename
$\mathcal{E}[X = expr]$	attribute extension and expression evaluation
$\mathcal{A}[X = a(A); B]$	attribute extension and aggregate function evaluation

Table 1: Additional algebraic operators

2.1.1 Not-Exists Semijoin

The *not-exists semijoin* operator, $\ltimes_{\nexists p}$, is introduced to concisely express *negative subqueries* as they are expressed in SQL (e.g. **not exists**); negative subqueries appear frequently in rule definitions [CW90]. The not-exists semijoin operator is defined as:

$$E_1 \ltimes_{\nexists p} E_2 = E_1 - (E_1 \ltimes_p E_2)$$

Note that we could instead define the relational difference operator in terms of not-exists semijoin: $E_1 - E_2 = E_1 \ltimes_{\nexists} E_2$ (with renaming of attributes in E_1 and E_2 as necessary). Hence, for convenience, we consider only the not-exists semijoin and not the difference operator in the remainder of the paper.

2.1.2 Aggregate Functions and Expression Evaluation

The *attribute extension* operators allow us to extend a relational expression E with a new attribute; this approach is used for aggregate functions and for modification operations. We have:

- The $\mathcal{E}$ operator, which computes expressions applied to each tuple of E
- The $\mathcal{A}$ operator, which computes aggregate functions (e.g. `max`, `min`, `avg`, `sum`, `count`) over partitions of E

$\mathcal{E}$ is a unary operator applied to a relational expression E producing a result with schema $schema(E) \cup \{X\}$. Recall from Table 1 that the $\mathcal{E}$ operator is expressed as:

$$\mathcal{E}[X = expr]E$$

expr is an expression evaluated over each tuple t of E (a conventional expression involving attributes of t and constants) yielding one value for each tuple; this value is entered into the new attribute X for each tuple of E. For details of similar operators see [CCRL$^+$90]; examples are given in later sections.

Operation	Algebraic expression	New database state
insert	E_{ins}	$R \cup E_{ins}$
delete	E_{del}	$R \ltimes_{\nexists} E_{del}$
update	E_{upd}	$(R \ltimes_{\nexists} E_{upd}) \ \cup \ \alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$

Table 2: Algebraic description of insert, delete, and update operations

$\mathcal{A}$ is also a unary operator applied to a relational expression E producing a result with schema $schema(E) \cup \{X\}$. Recall from Table 1 that the $\mathcal{A}$ operator is expressed as:

$$\mathcal{A}[X = a(A); B]E$$

B defines a set of attributes on which the result of E is partitioned; each group in the partition contains all the tuples with the same B value. a is an aggregate function that is applied to the (multiset of) values contained in the projection of each partition on attribute A, yielding one value for each partition; this value is entered into the new attribute X for each tuple of the partition. The attributes B are optional: when B is omitted, no grouping is performed, and the aggregate function a is applied to the entire result of E, yielding one value; that value is entered into the new attribute X for each tuple of E. For details see [CG85].

2.1.3 Modification Operations

We represent data modification operations in relational algebra by characterizing the operations in terms of the database state they produce. Table 2 presents inserts, deletes, and updates by indicating the algebraic expressions that are used to denote the operations, and the way in which these expressions are applied to a relation R to produce a new value for R. In the table, A_u denotes the attributes of R that are updated, A'_u denotes primed versions of these attributes (explained below), and $A_r = schema(R) - A_u$.

Insert operation. An insert operation is denoted by a relational expression E_{ins}. E_{ins} produces the tuples to be inserted (either a set of constant tuples or the result of an algebraic expression). The schema of E_{ins} must coincide with the schema of R.

Delete operation. A delete operation is denoted by a relational expression E_{del}. E_{del} produces the tuples to be deleted. The schema of E_{del} must coincide with the schema of R.

Update operation. An update operation is denoted by a relational expression E_{upd}. E_{upd} has schema $schema(R) \cup A'_u$, where attributes A'_u contain the new values for the updated attributes A_u. As convention, the new values for the updated attributes are always assigned the corresponding "primed" attribute names. That is, if attribute $A \in A_u$ is updated, then the new value for A is assigned to attribute A'.

A typical way to express E_{upd} is:

$$E_{upd} = \mathcal{E}[A'_{u1} = expr_1]\ \mathcal{E}[A'_{u2} = expr_2]\ \ldots\ \mathcal{E}[A'_{un} = expr_n]\ E_c$$

where E_c is an expression producing the tuples to be updated (i.e. the "selection condition" of the update operation). The schema of E_c must coincide with the schema of R. $\mathcal{E}[A'_{ui} = expr_i]$ evaluates expression $expr_i$ on each tuple of E_c and assigns the result to the new attribute A'_{ui}. Although this is a useful form, in its generality E_{upd} can be any relational expression with schema $schema(R) \cup A'_u$.

As specified in Table 2, the new state of R after the update operation is the union of two terms:

1. The first term $R \ltimes_{\nexists} E_{upd}$ includes in the result all tuples in R that are not modified by the update operation.

2. The second term $\alpha_{A'_u;A_u}(\pi_{A_r,A'_u} E_{upd})$ includes in the result the original values for the non-updated attributes of the modified tuples and the new values for the modified attributes, with the primed attribute names replaced by the original attribute names.

Given a relational expression E with schema $schema(R) \cup A'_u$, we often need the corresponding expression that is compatible in schema with R and contains either the pre-updated (old) or the updated (new) values for the modified attributes. For convenience we will use the abbreviations $\rho_{old}(E) = \pi_{schema(E)-A'_u} E$ and $\rho_{new}(E) = \alpha_{A'_u;A_u}(\pi_{schema(E)-A_u} E)$.

2.2 Rule Syntax and Semantics

A Condition-Action rule in our language is defined as:

$$E_{cond} \ \rightarrow\ E_{act}$$

where:

- E_{cond} states the rule's condition as an expression in our extended relational algebra.
- E_{act} states the rule's action as a data modification operation expressed using E_{ins}, E_{del}, or E_{upd} as given in Table 2.[1]

When this rule is evaluated, the condition E_{cond} is true if and only if $E_{cond} - E^{old}_{cond} \neq \emptyset$, where E^{old}_{cond} denotes the result of E_{cond} the last time the rule was evaluated during rule processing. If the rule has not previously been evaluated, then $E^{old}_{cond} = \emptyset$. That is, informally, the condition is true whenever the query produces "new" tuples. This is identical to the interpretation of conditions in the Condition-Action rules of, e.g., *Ariel* [Han92], *RPL* [DE89], and set-oriented adaptations

[1] For simplicity, we consider rules with a single action here, although many expert database systems allow rules with a sequence of actions. Our methods easily extend to multiple actions, usually simply by applying the method once for each action [Bar94].

of OPS5 [GP91]; it also is similar to the way many Event-Condition-Action rules appear to be programmed in practice [CW90].

The action E_{act} is a normal data modification operation executed on the current database state. In some expert database systems, e.g. [GP91,Han92], a rule's action implicitly operates only on the data "selected" by the condition, rather than on the entire database. We could use a similar rule model here, but it would complicate the syntax and semantics and has no bearing on our analysis methods; see Section 6 for further discussion.

Rule processing is invoked after some set of user or application modifications to the database. The basic algorithm for rule processing is:

```
repeat until no rule has a true condition:
   select a rule r with a true condition;
   execute r's action
```

In this paper, we do not consider the effect of a *conflict resolution policy* for selecting among multiple rules with true conditions [HW93]. However, as an extension to our framework we plan to incorporate conflict resolution using rule priorities; see Section 6. Note also that the "granularity" of rule processing invocation with respect to database modifications [HW93] is irrelevant here in the context of rule analysis.

2.3 Examples

In this section we give the algebraic representation of five rules. These rules will be used as examples throughout the paper. All five rules refer to the following relations:

```
ACCOUNT(num,name,balance,rate)
CUSTOMER(name,address,city)
LOW-ACC(num,name,date)
```

Relation `ACCOUNT` contains information on a bank's accounts, while relation `CUSTOMER` contains information on the bank's customers. Relation `LOW-ACC` contains all accounts with a low balance, including the date on which the balance became low. We assume that the first attribute is a key for each relation, although our method does not rely on this assumption.

Example 2.1: Rule `bad-account` states that if an account has a balance less than 500 and an interest rate greater than 0%, then that account's interest rate is set to 0%. In our language this rule is expressed as:

$$E_{cond} \rightarrow E_{upd}$$

where

$$\begin{aligned} E_{cond} &= \pi_{\texttt{balance},\texttt{rate}}(\sigma_{\texttt{balance}<500 \wedge \texttt{rate}>0}\texttt{ACCOUNT}) \\ E_{upd} &= \mathcal{E}[\texttt{rate}' = 0]E_c \\ E_c &= \sigma_{\texttt{balance}<500 \wedge \texttt{rate}>0}\texttt{ACCOUNT} \end{aligned}$$

Example 2.2: Rule `raise-rate` states that if an account has an interest rate greater than 1% but less than 2%, then the interest rate is raised to 2%. In our language this rule is expressed as:

$$E_{cond} \rightarrow E_{upd}$$

where

$$\begin{array}{lcl} E_{cond} & = & \pi_{\mathtt{rate}}(\sigma_{\mathtt{rate}>1\, \wedge \mathtt{rate}<2}\mathtt{ACCOUNT}) \\ E_{upd} & = & \mathcal{E}[\mathtt{rate}' = 2]E_c \\ E_c & = & \sigma_{\mathtt{rate}>1\, \wedge \mathtt{rate}<2}\mathtt{ACCOUNT} \end{array}$$

Example 2.3: Rule `SF-bonus` states that when the number of customers living in San Francisco exceeds 1000, then the interest rate of all San Francisco customers' accounts with a balance greater than 5000 and an interest rate less than 3% is increased by 1%. In our language this rule is expressed as:

$$E_{cond} \rightarrow E_{upd}$$

where

$$\begin{array}{lcl} E_{cond} & = & \pi_{\mathtt{city,C}}(\sigma_{\mathtt{C}>1000}(\mathcal{A}[\mathtt{C} = \mathtt{count(name)}](\sigma_{\mathtt{city='SF'}}\mathtt{CUSTOMER}))) \\ E_{upd} & = & \mathcal{E}[\mathtt{rate}' = \mathtt{rate} + 1]E_c \\ E_c & = & (\sigma_{\mathtt{balance}>5000\, \wedge \mathtt{rate}<3}\mathtt{ACCOUNT}) \ltimes_{name} (\sigma_{\mathtt{city='SF'}}\mathtt{CUSTOMER}) \end{array}$$

and *name* is an abbreviation for `ACCOUNT.name = CUSTOMER.name`.

Example 2.4: Rule `add-to-bad` states that if an account has a balance less than 500 and is not yet recorded in the `LOW-ACC` relation, then the information on that account is inserted into the `LOW-ACC` relation, "time-stamped" with the current date. In our language this rule is expressed as:

$$E_{cond} \rightarrow E_{ins}$$

where

$$\begin{array}{lcl} E_{cond} & = & \pi_{\mathtt{num,balance}}((\sigma_{\mathtt{balance}<500}\mathtt{ACCOUNT}) \ltimes_{\nexists num} \mathtt{LOW\text{-}ACC}) \\ E_{ins} & = & \mathcal{E}[\mathtt{date} = \mathtt{today()}]\pi_{\mathtt{num,name}}((\sigma_{\mathtt{balance}<500}\mathtt{ACCOUNT}) \ltimes_{\nexists num} \mathtt{LOW\text{-}ACC}) \end{array}$$

and *num* is an abbreviation for `ACCOUNT.num = LOW-ACC.num` and `today()` is a system defined function returning the current date.

Example 2.5: Rule `delete-from-bad` states that if an account in the `LOW-ACC` relation has a balance of at least 500 in the `ACCOUNT` relation, then the account is deleted from the `LOW-ACC` relation. In our language this rule is expressed as:

$$E_{cond} \rightarrow E_{del}$$

where

$$\begin{array}{lcl} E_{cond} & = & \pi_{\mathtt{num}}(\mathtt{LOW\text{-}ACC} \ltimes_{num} (\sigma_{\mathtt{balance}\geq 500}\mathtt{ACCOUNT})) \\ E_{del} & = & \mathtt{LOW\text{-}ACC} \ltimes_{num} (\sigma_{\mathtt{balance}\geq 500}\mathtt{ACCOUNT}) \end{array}$$

and *num* is an abbreviation for `LOW-ACC.num = ACCOUNT.num`.

3 The Propagation Algorithm

We describe a general algorithm that uses syntactic analysis to predict how a database query (i.e. a rule condition) can be affected by the execution of a data modification operation (i.e. a rule action). The outcome of our *Propagation Algorithm* is zero or more of the operations *insert*, *delete*, and *update*, characterizing how the result of the query may change due to the execution of the modification: If the algorithm produces an *insert* operation, then the query may contain more data after the modification; if the algorithm produces a *delete* operation, then the query may contain less data after the modification; if the algorithm produces an *update* operation, then the query may contain updated data after the modification; if no operations are produced, then the result of the query cannot change due to the modification. The operations produced by our algorithm are represented as relational expressions in the same way that we algebraically represent data modification operations in rule actions, except here the modifications apply to arbitrary relational expressions instead of only to single relations.

The algorithm takes as input a rule condition C and a rule action A, both expressed in extended relational algebra as defined in Section 2. As an initial filter, if the condition C does not reference the relation modified by A, then clearly A cannot affect the result of C. Otherwise, A is "propagated" through a tree representation of C's query. The leaves of the tree are relations, and one of these leaves corresponds to the relation R that is modified by A. (We assume there is only one reference to R in condition C; our method can easily be extended to handle multiple references [Bar94].) Action A is propagated from the affected relation up the query tree, and it may be transformed into one or more different actions (modification operations) during the propagation process. To describe the propagation, we give formal rules specifying how arbitrary actions are propagated through arbitrary nodes of the tree. After each propagation through a node in the tree, the actions obtained are checked for "consistency" (explained next). Inconsistent actions are discarded, while consistent actions are further propagated. The propagation process continues until the root of the query tree is reached or all actions have been discarded as inconsistent. At each point during the propagation process, the actions associated with a node N in the tree indicate the actions that may occur to N's subtree as a result of performing the original action A. Hence, the consistent actions that reach the root of the tree describe how the original action A may affect condition C.

An action produced by the propagation process is *consistent* when the algebraic expression describing the action does not contain contradictions, i.e. it is satisfiable. Satisfiability of relational expressions is undecidable in the general case, so we can give sufficient but not necessary conditions for satisfiability of the expressions representing the propagated actions. However, for most expressions that arise in practice, either we can see trivially whether the expression is satisfiable (as in examples below), or we can verify satisfiability using the tableau method in [Ull89].[2]

Figure 1 illustrates the propagation of an insert action (described by expression E_{ins}) on relation

[2]Note that a "conservative" test for satisfiability is not really a limitation here, since our entire approach is based on syntactic analysis and hence is conservative: when an expression is satisfied we determine only that the condition *may* be affected, not that the condition necessarily will be affected.

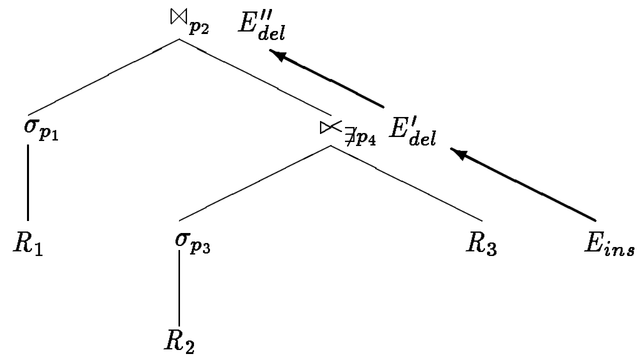

Figure 1: Propagation of E_{ins} action

R_3 through the nodes of the query tree representing the condition $C = (\sigma_{p_1} R_1) \bowtie_{p_2} (\sigma_{p_3} R_2 \ltimes_{\not\exists p_4} R_3)$. The bold line represents the propagation path of the E_{ins} action: The E_{ins} action is first substituted for the affected relation R_3. Then, starting from the $\ltimes_{\not\exists p_4}$ node, for each node with an operand affected by the E_{ins} action, the corresponding propagated expression is computed. At the end of the propagation process a delete action E''_{del} is obtained at the root. As the reader may verify, an insert operation on R_3 may only cause data satisfying C to be deleted.

The rules for propagation are given in tables based on the kind of incoming action: insert and delete actions in Tables 3 and 4 respectively, and update actions in Tables 5 and 6. Each row in the tables contains the propagated action(s), E^{out}, as a function of the incoming action, E^{in}, and the relational operator in the query tree. The column labeled "Applicability condition" specifies when different propagation rules are used for different cases. In the tables, A_1, A_2, and B are attribute lists, $A_{jn} = schema(E_1) \cap schema(E_2)$, $A_{E_2} = schema(E_2)$, A_u are the updated attributes, A_p and A_e are the attributes involved in predicate p and expression $expr$ respectively, $p' = \alpha_{A_u;A'_u} p$ and $expr' = \alpha_{A_u;A'_u} expr$, $p(B)$ equates all attributes in list B, and $p'(A_{uB})$ equates all attributes in A'_{uB} with the corresponding B attributes. Since the natural join, cartesian product, and union operators are symmetric, without loss of generality we assume that the first operand is modified; analogous rules apply for modifications to the second operand. Observe that aggregate functions require, in addition to the incoming action, the entire relational expression E to which the aggregate function is applied.

The formulas given in Table 5 don't take into account the internal structure of selection predicates and update expressions. In the case of simple predicates (comparisons between an attribute and a constant[3]) and simple arithmetic update expressions (addition or subtraction of constants from an attribute), in many cases it is possible to eliminate some of the propagated actions. For example, consider the propagation of the update $A = A+1$ through the operation $\sigma_{A>5}$. Intuitively,

[3]Actually, any expression involving non-updated attributes and constants can be considered as a constant in this context.

	Propagated action: $E^{in}_{ins} \rightarrow E^{out}_{action}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$		$E^{out}_{ins} = \sigma_p E^{in}_{ins}$
$\pi_{A_1} E$		$E^{out}_{ins} = \pi_{A_1} E^{in}_{ins}$
$E_1 \bowtie E_2$		$E^{out}_{ins} = E^{in}_{ins} \bowtie E_2$
$E_1 \times E_2$		$E^{out}_{ins} = E^{in}_{ins} \times E_2$
$E_1 \cup E_2$		$E^{out}_{ins} = E^{in}_{ins} \ltimes_{\nexists} E_2$
$E_1 \ltimes_p E_2$	insert into E_1	$E^{out}_{ins} = E^{in}_{ins} \ltimes_p E_2$
	insert into E_2	$E^{out}_{ins} = E_1 \ltimes_p E^{in}_{ins}$
$E_1 \ltimes_{\nexists p} E_2$	insert into E_1	$E^{out}_{ins} = E^{in}_{ins} \ltimes_{\nexists p} E_2$
	insert into E_2	$E^{out}_{del} = E_1 \ltimes_p E^{in}_{ins}$
$\alpha_{A_1;A_2} E$		$E^{out}_{ins} = \alpha_{A_1;A_2} E^{in}_{ins}$
$\mathcal{E}[X = expr]E$		$E^{out}_{ins} = \mathcal{E}[X = expr] E^{in}_{ins}$
$\mathcal{A}[X = a(A); B]E$	$B = \emptyset$	$E^{out}_{ins} = E^{in}_{ins} \bowtie \mathcal{A}[X = a(A)](E \cup E^{in}_{ins})$ $E^{out}_{upd} = (\mathcal{A}[X' = a(A)](E \cup E^{in}_{ins})) \bowtie (\mathcal{A}[X = a(A)]E)$
	$B \neq \emptyset$	$E^{out}_{ins} = E^{in}_{ins} \bowtie \mathcal{A}[X = a(A); B](E \cup E^{in}_{ins})$ $E^{out}_{upd} = ((\mathcal{A}[X' = a(A); B](E \cup E^{in}_{ins})) \bowtie (\mathcal{A}[X = a(A); B]E)) \ltimes_{p(B)} E^{in}_{ins}$

Table 3: Insert action propagation

	Propagated action: $E^{in}_{del} \rightarrow E^{out}_{action}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$		$E^{out}_{del} = \sigma_p E^{in}_{del}$
$\pi_{A_1} E$		$E^{out}_{del} = \pi_{A_1} E^{in}_{del}$
$E_1 \bowtie E_2$		$E^{out}_{del} = E^{in}_{del} \bowtie E_2$
$E_1 \times E_2$		$E^{out}_{del} = E^{in}_{del} \times E_2$
$E_1 \cup E_2$		$E^{out}_{del} = E^{in}_{del} \ltimes_{\nexists} E_2$
$E_1 \ltimes_p E_2$	delete from E_1	$E^{out}_{del} = E^{in}_{del} \ltimes_p E_2$
	delete from E_2	$E^{out}_{del} = E_1 \ltimes_p E^{in}_{del}$
$E_1 \ltimes_{\nexists p} E_2$	delete from E_1	$E^{out}_{del} = E^{in}_{del} \ltimes_{\nexists p} E_2$
	delete from E_2	$E^{out}_{ins} = E_1 \ltimes_p E^{in}_{del}$
$\alpha_{A_1;A_2} E$		$E^{out}_{del} = \alpha_{A_1;A_2} E^{in}_{del}$
$\mathcal{E}[X = expr]E$		$E^{out}_{del} = \mathcal{E}[X = expr] E^{in}_{del}$
$\mathcal{A}[X = a(A); B]E$	$B = \emptyset$	$E^{out}_{del} = E^{in}_{del} \bowtie (\mathcal{A}[X = a(A)]E)$ $E^{out}_{upd} = (\mathcal{A}[X' = a(A)](E \ltimes_{\nexists} E^{in}_{del})) \bowtie (\mathcal{A}[X = a(A)]E)$
	$B \neq \emptyset$	$E^{out}_{del} = E^{in}_{del} \bowtie \mathcal{A}[X = a(A); B]E$ $E^{out}_{upd} = ((\mathcal{A}[X' = a(A); B](E \ltimes_{\nexists} E^{in}_{del})) \bowtie (\mathcal{A}[X = a(A); B]E)) \ltimes_{p(B)} E^{in}_{del}$

Table 4: Delete action propagation

	Propagated action: $E^{in}_{upd} \rightarrow E^{out}_{action}$	
Node	Applicability condition	Resulting expression
$\sigma_p E$	$A_u \cap A_p = \emptyset$	$E^{out}_{upd} = \sigma_p E^{in}_{upd}$
	$A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((\sigma_{p'} E^{in}_{upd}) \ltimes_{\nexists} (\sigma_p E^{in}_{upd}))$ $E^{out}_{del} = \rho_{old}((\sigma_p E^{in}_{upd}) \ltimes_{\nexists} (\sigma_{p'} E^{in}_{upd}))$ $E^{out}_{upd} = (\sigma_{p'} E^{in}_{upd}) \bowtie (\sigma_p E^{in}_{upd})$
$\pi_{A_1} E$	$A_u \cap A_1 = \emptyset$	$\emptyset$
	$A_u \cap A_1 = A_{u1}$	$E^{out}_{upd} = \pi_{A_1, A'_{u1}} E^{in}_{upd}$
$E_1 \bowtie E_2$	$A_u \cap A_{jn} = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \bowtie E_2$
	$A_u \cap A_{jn} = A_{ujn}$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \bowtie (\alpha_{A_{ujn};A'_{ujn}} E_2)) \ltimes_{\nexists} (E^{in}_{upd} \bowtie E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \bowtie E_2) \ltimes_{\nexists} (E^{in}_{upd} \bowtie (\alpha_{A_{ujn};A'_{ujn}} E_2)))$ $E^{out}_{upd} = ((E^{in}_{upd} \bowtie (\alpha_{A_{E_2};A'_{E_2}} E_2)) \bowtie (E^{in}_{upd} \bowtie E_2))$
$E_1 \times E_2$		$E^{out}_{upd} = E^{in}_{upd} \times E_2$
$E_1 \cup E_2$		$E^{ins}_{out} = \rho_{new}((E^{in}_{upd} \ltimes E_2) \ltimes_{\nexists} (\alpha_{A_u;A'_u} E_2))$ $E^{del}_{out} = \rho_{old}((E^{in}_{upd} \ltimes_{\nexists} E_2) \ltimes (\alpha_{A_u;A'_u} E_2))$ $E^{upd}_{out} = (E^{in}_{upd} \ltimes_{\nexists} E_2) \ltimes_{\nexists} (\alpha_{A_u;A'_u} E_2)$
$E_1 \ltimes_p E_2$	update E_1, $A_u \cap A_p = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \ltimes_p E_2$
	update E_1, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \ltimes_{p'} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_p E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \ltimes_p E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{p'} E_2))$ $E^{out}_{upd} = (E^{in}_{upd} \ltimes_{p'} E_2) \bowtie (E^{in}_{upd} \ltimes_p E_2)$
	update E_2, $A_u \cap A_p = \emptyset$	$\emptyset$
	update E_2, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = (E_1 \ltimes_{p'} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_p E^{in}_{upd})$ $E^{out}_{del} = (E_1 \ltimes_p E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{p'} E^{in}_{upd})$
$E_1 \ltimes_{\nexists p} E_2$	update E_1, $A_u \cap A_p = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \ltimes_{\nexists p} E_2$
	update E_1, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = \rho_{new}((E^{in}_{upd} \ltimes_{\nexists p'} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{\nexists p} E_2))$ $E^{out}_{del} = \rho_{old}((E^{in}_{upd} \ltimes_{\nexists p} E_2) \ltimes_{\nexists} (E^{in}_{upd} \ltimes_{\nexists p'} E_2))$ $E^{out}_{upd} = (E^{in}_{upd} \ltimes_{\nexists p'} E_2) \bowtie (E^{in}_{upd} \ltimes_{\nexists p} E_2)$
	update E_2, $A_u \cap A_p = \emptyset$	$\emptyset$
	update E_2, $A_u \cap A_p \neq \emptyset$	$E^{out}_{ins} = (E_1 \ltimes_{\nexists p'} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{\nexists p} E^{in}_{upd})$ $E^{out}_{del} = (E_1 \ltimes_{\nexists p} E^{in}_{upd}) \ltimes_{\nexists} (E_1 \ltimes_{\nexists p'} E^{in}_{upd})$
$\alpha_{A_1;A_2} E$	$A_1 \cap A_u = \emptyset$	$E^{out}_{upd} = \alpha_{A_1;A_2} E^{in}_{upd}$
	$A_1 \cap A_u = A_{u1}$,	$E^{out}_{upd} = \alpha_{A_1;A_2} \alpha_{A'_{u1};A'_{u2}} E^{in}_{upd}$
$\mathcal{E}[X = expr]E$	$A_u \cap A_e = \emptyset$	$E^{out}_{upd} = \mathcal{E}[X = expr] E^{in}_{upd}$
	$A_u \cap A_e \neq \emptyset$	$E^{out}_{upd} = \mathcal{E}[X = expr] \mathcal{E}[X' = expr'] E^{in}_{upd}$

Table 5: Update action propagation

		Propagated action: $E^{in}_{upd} \rightarrow E^{out}_{action}$
Node	Applicability condition	Resulting expression
$\mathcal{A}[X = a(A); B]E$	$A_u \cap A = \emptyset$, $A_u \cap B = \emptyset$	$E^{out}_{upd} = E^{in}_{upd} \bowtie (\mathcal{A}[X = a(A); B]E)$
	$B = \emptyset$, $A_u \supseteq A$	$E^{out}_{upd1} = E^{in}_{upd} \bowtie ((\mathcal{A}[X = a(A)]E) \bowtie (\mathcal{A}[X' = a(A)](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd}))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A)]E) \bowtie (\mathcal{A}[X' = a(A)](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd})))) \ltimes_{\nexists} E^{in}_{upd}$
	$B \neq \emptyset$, $A_u \supseteq A$, $A_u \cap B = \emptyset$	$E^{out}_{upd1} = E^{in}_{upd} \bowtie ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd}))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd})))) \ltimes_{p(B)} E^{in}_{upd}) \ltimes_{\nexists} E^{in}_{upd}$
	$B \neq \emptyset$, $A_u \cap B = A_{uB}$	$E^{out}_{upd1} = E^{in}_{upd} \bowtie ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd}))))$ $E^{out}_{upd2} = ((\mathcal{A}[X = a(A); B]E) \bowtie (\mathcal{A}[X' = a(A); B](\rho_{new}(E^{in}_{upd}) \cup (E \ltimes_{\nexists} E^{in}_{upd}))) \ltimes_{p(B) \wedge p'(A_{uB})} E^{in}_{upd}) \ltimes_{\nexists} E^{in}_{upd}$

Table 6: Update action propagation (cont.)

	Arithmetic update expression		
predicate	addition	subtraction	other
$A_u = k$, $A_u \neq k$	E_{upd}	E_{upd}	E_{upd}
$A_u > k$, $A_u \geq k$	E_{del}	E_{ins}	–
$A_u < k$, $A_u \leq k$	E_{ins}	E_{del}	–

Table 7: Eliminated actions

this update will never cause tuples to be deleted from the expression rooted in $\sigma_{A>5}$. Thus, the propagated delete operation can be eliminated. Table 7 shows the actions that can be eliminated in the different cases. In the table, "other" indicates an arbitrary arithmetic expression, which in the case of an equality or non-equality predicate still allows an update action to be eliminated.

3.1 Examples

We give two examples of the Propagation Algorithm applied to rules from Section 2.3. In each example, we analyze the effect of one rule's action on another rule's condition by fully describing the propagation process and the satisfiability test.

Example 3.1: Consider condition E_{cond} in rule `bad-account` (Example 2.1) and the update action in rule `SF-bonus` (Example 2.3). The input to the algorithm is:

$$C = \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}\texttt{ACCOUNT})$$
$$A = E_{upd} = \mathcal{E}[\texttt{rate}' = \texttt{rate} + 1](\sigma_{\texttt{balance>5000}\,\wedge\texttt{rate<3}}(\texttt{ACCOUNT} \ltimes_{name} \textit{SF-cust}))$$

where *name* is an abbreviation for `ACCOUNT.name = CUSTOMER.name` and *SF-cust* is an abbreviation for $\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER}$. Using Table 5, the propagation of E_{upd} through the selection operation in C yields insert and update actions (the delete action is eliminated, see Table 7). We have:

$$E'_{ins} = \rho_{new}((\sigma_{\texttt{balance<500}\,\wedge\texttt{rate'>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}E_{upd}))$$
$$E'_{upd} = (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate'>0}}E_{upd}) \bowtie (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}E_{upd})$$

In both cases, predicates `balance < 500` and `balance > 5000` (the latter from E_{upd}) are contradictory, so both expressions E'_{ins} and E'_{upd} are unsatisfiable. Intuitively, action A operates on data not read by condition C. We conclude that action A cannot affect condition C.

Example 3.2: Consider condition E_{cond} in rule `bad-account` (Example 2.1) and the update action in rule `raise-rate` (Example 2.2). The input to the algorithm is:

$$C = \pi_{\texttt{balance,rate}}(\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}\texttt{ACCOUNT})$$
$$A = E_{upd} = \mathcal{E}[\texttt{rate}' = 2](\sigma_{\texttt{rate>1}\,\wedge\texttt{rate<2}}\texttt{ACCOUNT})$$

The propagation of E_{upd} through the selection operation in C yields insert, delete, and update actions:

$$E'_{ins} = \rho_{new}((\sigma_{\texttt{balance<500}\,\wedge\texttt{rate'>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}E_{upd}))$$
$$E'_{del} = \rho_{old}((\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate'>0}}E_{upd}))$$
$$E'_{upd} = (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate'>0}}E_{upd}) \bowtie (\sigma_{\texttt{balance<500}\,\wedge\texttt{rate>0}}E_{upd})$$

These expressions do not contain contradictory predicates, thus they may be satisfiable and the propagation continues. The propagation of E'_{ins}, E'_{del}, and E'_{upd} through the projection operation in C yields:

$$E''_{ins} = \pi_{\texttt{balance,rate}}E'_{ins}$$
$$E''_{del} = \pi_{\texttt{balance,rate}}E'_{del}$$
$$E''_{upd} = \pi_{\texttt{balance,rate,rate'}}E'_{upd}$$

All three expressions are satisfiable, thus action A can affect the result of condition C. Furthermore, E''_{ins}, E''_{del}, and E''_{upd} describe the actions that can be performed on C as a result of the execution of A.

3.2 Correctness of the Algorithm

The following Theorem states the correctness of the Propagation Algorithm. The proof proceeds step-by-step for each propagation rule given in Tables 3–6, and the proof technique is analogous for all rules. Hence, due to space constraints, we outline the proof procedure for only one propagation rule; see [Bar94] for a complete proof.

Theorem 3.1: Let Q be the query tree corresponding to a relational expression C and let A be an action performed on a relation in Q. Let E^{out} be the actions produced at the root of Q by application of the propagation rules in Tables 3–6. E^{out} describes a superset of all actions that can be performed on expression C as a result of executing the original action A.

Proof sketch: The proof proceeds by induction on the depth of Q. *Base case:* Q is a single node representing the modified relation. Then $E^{out} = A$; correctness in this case is obvious. *Induction step:* Let the root of Q be a unary (resp. binary) relational operator *op* over a subtree S with incoming actions E^{in} (resp. over two subtrees S and S', of which S has incoming actions E^{in}). By the induction hypothesis, we assume E^{in} describes a superset of the actions that are performed on the expression rooted in S as a result of executing the original action A. We must show that, if we apply the appropriate propagation rule for *op* to obtain E^{out} from E^{in}, then E^{out} describes a superset of the actions that are performed on the expression rooted in Q as a result of executing the original action A. As an example, let *op* be a selection σ_p performed over an arbitrary subtree S and consider an update action E^{in}_{upd} associated with S and performed on an attribute in p. Applying our propagation rules from the second line of Table 5, we obtain a triple $\langle E^{out}_{ins}, E^{out}_{del}, E^{out}_{upd}\rangle$, corresponding to tuples added to, deleted from, and updated in the result of $Q = \sigma_p S$. It can be seen that $E^{out}_{ins} = \rho_{new}((\sigma_{p'} E^{in}_{upd}) \ltimes_{\nexists} (\sigma_p E^{in}_{upd}))$ describes tuples that now satisfy the selection predicate as a result of E^{in}_{upd}, hence they are added to the result of Q. Analogously, it can be seen that $E^{out}_{del} = \rho_{old}((\sigma_p E^{in}_{upd}) \ltimes_{\nexists} (\sigma_{p'} E^{in}_{upd}))$ describes tuples that now do not satisfy the selection predicate as a result of E^{in}_{upd}, hence they are deleted from the result of Q. Finally, $E^{out}_{upd} = (\sigma_{p'} E^{in}_{upd}) \bowtie (\sigma_p E^{in}_{upd})$ describes the tuples that satisfy the selection predicate before and after E^{in}_{upd} but with different values, hence they are updated in the result of Q. The other propagation rules are verified similarly. □

4 Termination Analysis

Recall the rule processing loop from Section 2.2. Termination for a rule set is guaranteed if rule processing always reaches a state in which no rule has a true condition. Notice that, according to the semantics in Section 2.2, after the first execution of each rule r, r's condition is true again if and only if new data satisfies the condition. Hence, informally, rule processing does not terminate if and only if rules provide new data to each other indefinitely.

We say that a rule r_1 *may activate* a rule r_2 if executing r_1's action may cause new data to satisfy r_2's condition. We analyze termination by building an *Activation Graph*. In the graph, nodes represent rules, and directed edges indicate that one rule may activate the other. If there are no cycles in the graph, then rule processing is guaranteed to terminate [BCW93,AWH92]. Hence, the core of termination analysis is determining when an edge should be included in the graph, i.e. when one rule may activate another rule. The more accurately we can make this decision, the more accurately we can analyze termination.

We use our Propagation Algorithm to decide when an edge $r_i \rightarrow r_j$ belongs in the Activation

Graph. Note that rules may activate themselves, so $r_i = r_j$ is included in the analysis. To detect if r_i may activate r_j, the Propagation Algorithm is applied to r_j's condition C and r_i's action A. If the algorithm yields an *insert* or *update* operation, then r_i may provide new data satisfying r_j's condition. Thus, r_i may activate r_j, and the edge $r_i \to r_j$ belongs in the graph. If only a *delete* operation or no operation is produced by the algorithm, then r_i cannot provide new data to r_j's condition, and the edge is not included in the graph.

Our use of the Activation Graph is similar to, e.g., [AWH92,CW90], but our approach is far less conservative since we exploit the algebraic structure of conditions and actions to accurately determine when edges belong in the graph.

4.1 Examples

Consider the rules from Section 2.3. We present two examples where we apply our analysis techniques to determine that a pair of rules does not produce a cycle in the Activation Graph, i.e. the rules cannot activate each other indefinitely. In both of these examples, the technique in [AWH92] is unable to determine that these rules terminate.

Example 4.1: Consider rule `bad-account` (Example 2.1) and rule `raise-rate` (Example 2.2) that here will be called r_1 and r_2 respectively. Both rule conditions reference attribute `rate`, and both rule actions update `rate`. Hence, intuitively (and according to the method in [AWH92]), the two rules might activate each other indefinitely. We have shown in Example 3.2 that r_2's action may provide data to r_1's condition (since *insert* and *update* operations are produced by the Propagation Algorithm), thus the edge $r_2 \to r_1$ belongs in the Activation Graph. Now we use the Propagation Algorithm to determine if r_1 may activate r_2. The input to the algorithm is:

$$\begin{aligned} C &= \pi_{\texttt{rate}}(\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}\texttt{ACCOUNT}) \\ A &= E_{upd} = \mathcal{E}[\texttt{rate}' = 0](\sigma_{\texttt{balance}<500 \wedge \texttt{rate}>0}\texttt{ACCOUNT}) \end{aligned}$$

The propagation of E_{upd} through the selection operation in C yields:

$$\begin{aligned} E'_{ins} &= \rho_{new}((\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd})) \\ E'_{upd} &= (\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd}) \bowtie (\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd}) \\ E'_{del} &= \rho_{old}((\sigma_{\texttt{rate}>1 \wedge \texttt{rate}<2}E_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{rate}'>1 \wedge \texttt{rate}'<2}E_{upd})) \end{aligned}$$

Since predicates `rate' > 1` and `rate' = 0` (the latter from E_{upd}) are contradictory, expressions E'_{ins} and E'_{upd} are not satisfiable and hence are discarded. The propagation of E'_{del} through the projection operation in C yields:

$$E''_{del} = \pi_{\texttt{rate}} E'_{del}$$

which is satisfiable. Thus, r_1's action may result in a deletion of tuples from r_2's condition. However, since neither an *insert* nor an *update* action is produced, r_1 cannot activate r_2, the edge $r_1 \to r_2$ is not included in the Activation Graph, and we conclude that rules r_1 and r_2 will always terminate.

Example 4.2: Consider rule `add-to-bad` (Example 2.4) and rule `delete-from-bad` (Example 2.5) that here will be called r_1 and r_2 respectively. Here again, intuitively (and according to the method in [AWH92]), the two rules might activate each other indefinitely. We use the Propagation Algorithm to determine if r_1 may activate r_2. The input to the algorithm is:

$$C = \pi_{\text{num}}(\texttt{LOW-ACC} \ltimes_{num} (\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT}))$$
$$A = E_{ins} = \mathcal{E}[\texttt{date} = \texttt{today()}](\pi_{\text{num,name}}((\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}) \ltimes_{\not\exists num} \texttt{LOW-ACC}))$$

where *num* is an abbreviation for `LOW-ACC.num = ACCOUNT.num`. The propagation of E_{ins} through the semijoin operation in C yields:

$$E'_{ins} \quad = \quad (\mathcal{E}[\texttt{date} = \texttt{today()}](\pi_{\text{num,name}}(\textit{low-bal} \ltimes_{\not\exists num} \texttt{LOW-ACC}))) \ltimes_{num} \textit{high-bal}$$

where *low-bal* is an abbreviation for $\sigma_{\texttt{balance}<500}\texttt{ACCOUNT}$ and *high-bal* is an abbreviation for $\sigma_{\texttt{balance}\geq 500}\texttt{ACCOUNT}$. This expression is not satisfiable, since it requires a tuple with a given num value to satisfy both predicates $\texttt{balance} < 500$ and $\texttt{balance} \geq 500$. Hence, r_1 cannot activate r_2, edge $r_1 \rightarrow r_2$ is not included in the Activation Graph, and rules r_1 and r_2 are guaranteed to terminate.

5 Confluence Analysis

Recall again the rule processing loop from Section 2.2. In each iteration, there may be multiple rules eligible for execution, since more than one rule may have a true condition. A rule set is confluent if the final state of the database does not depend on which eligible rule is chosen for execution at any iteration.

To formally describe confluence and confluence analysis, we introduce the notion of a *rule execution state* and a *rule execution sequence*. Let R be the set of rules under consideration.

Definition 5.1: A *rule execution state* S is a pair (db, R_A), where db is a state of the database and $R_A \subseteq R$ is a set of activated rules. □

Definition 5.2: A *rule execution sequence* is a sequence σ consisting of a series of rule execution states linked by (executed) rules. A rule execution sequence is *complete* if the last state is $(db, \emptyset)$, i.e. the last state has no activated rules. A rule execution sequence is *valid* if it represents a correct execution sequence: only activated rules are executed, and pairs of adjacent states properly represent the effect of executing the corresponding rule; for details see [AWH92,Bar94]. □

We now define confluence in terms of execution sequences.

Definition 5.3: A rule set is *confluent* if, for every initial rule execution state S (corresponding to an initial database and set of modifications), every valid and complete rule execution sequence beginning with S has the same final state. □

Clearly we cannot use this definition directly to analyze confluence, since it requires the exhaustive verification of all possible execution sequences for all possible initial states. We give sufficient conditions for confluence based on the *commutativity* of rule pairs. Two rules r_i and r_j commute if, starting with any execution state S, executing r_i followed by r_j produces the same execution state as executing r_j followed by r_i. The conditions for commutativity require that the two rules cannot activate or deactivate each other,[4] and their actions can be executed in either order:

Definition 5.4: Distinct rules r_i and r_j commute if: (1) r_i's action cannot affect the outcome of r_j's condition (i.e. r_i can neither activate nor deactivate r_j); (2) executing r_i's action cannot change the effect of executing r_j's action; (3) conditions (1) and (2) with i and j reversed. □

Note that even though conditions (1)–(3) are not necessarily satisfied when $r_i = r_j$, it is the case that a rule always commutes with itself.

We prove two Lemmas, followed by the main Theorem on confluence. The first Lemma states, under the assumption of commutative rules, that two execution sequences with the same initial state and executed rules have the same final state; the second Lemma states, again under the assumption of commutativity, that two sequences with the same initial state must have the same executed rules.

Lemma 5.1: Let all pairs of rules in R commute. Let σ_1 and σ_2 be two valid and complete rule execution sequences with the same initial state, such that the same rules are executed in σ_1 and σ_2 although not necessarily in the same order. Then σ_1 and σ_2 have the same final state.

Proof: Since σ_1 and σ_2 have the same executed rules, we can "permute" σ_1 so that its rules are considered in the same order as σ_2. We exchange adjacent rules in σ_1 one pair at a time; with each exchange, there is no change to the outer two execution states due to commutativity. Hence, since σ_1 and σ_2 have the same initial state, they must have the same final state. □

Lemma 5.2: Let all pairs of rules in R commute. Let σ_1 and σ_2 be two valid and complete rule execution sequences with the same initial state. Then the same rules are executed in σ_1 and σ_2.

Proof: We again use commutativity to permute sequences without affecting outer execution states. In each sequence, we exchange rules one pair at a time until the rules appear in "sorted" order according to some criterion (the criterion is irrelevant as long as the same criterion is used for σ_1 and σ_2). Suppose, for the sake of a contradiction, that σ_1 and σ_2 have different executed rules, and consider the first point of divergence, i.e. where a rule r appears in σ_1 but a different rule r' appears in σ_2. Let S be the execution state preceding these rules; S is the same in σ_1 and σ_2, so r and r' are both activated in S. Without loss of generality, assume that r precedes r' in the sorted order. Then r cannot appear in σ_2 beyond S. Consequently, execution of some rule other than r in σ_2 must deactivate r. But this contradicts condition (1) of commutativity. □

---

[4]Rule r_i *deactivates* rule r_j if r_i's action deletes all new data satisfying r_i's condition.

Based on these Lemmas, the following Theorem presents a sufficient condition to guarantee confluence of a rule set.

Theorem 5.1: A rule set R is confluent if all pairs of rules in R commute.

Proof: Suppose all rule pairs commute, and consider two valid and complete execution sequences σ_1 and σ_2 with the same initial state. By Lemma 5.2, σ_1 and σ_2 have the same set of executed rules. Then by Lemma 5.1, σ_1 and σ_2 have the same final state. $\square$

The requirement for confluence in Theorem 5.1 may seem rather strong, but there is no way to weaken this requirement in a rule model without a more sophisticated conflict resolution policy or priorities among rules. We believe this argues for the importance of rule priorities, which we plan to investigate in this context as future work. Notice also that, in the case where no rule can activate itself, the confluence requirement as stated in Theorem 5.1 trivially implies termination, since the pairwise commutativity of all rules includes the requirement that no rule activates another rule. However, if one or more rules can activate themselves, then confluence does not imply termination.

Commutativity of rule pairs forms the basis of most methods for analyzing confluence of database rules, e.g. [AWH92,vdVS93]. The remainder of this section describes our technique for determining commutativity of rule pairs. Since commutativity itself is a "subroutine" to proving confluence, our commutativity analysis technique also can be applied in other contexts, e.g. [AWH92]. Needless to say, we use our Propagation Algorithm to analyze commutativity, exploiting the algebraic description of rule conditions and actions to yield a much more accurate analysis technique than, e.g., [AWH92].

To guarantee commutativity of two rules r_i and r_j, we must verify conditions (1), (2), and (3) in Definition 5.4. For (1), we determine that r_i cannot activate r_j exactly as we have done for termination; recall Section 4. To show that r_i cannot deactivate r_j, we must show that r_i's action A cannot "take away" data from r_j's condition C. It is easy to see that action A can take away data from condition C only if the Propagation Algorithm applied to A and C produces a *delete* operation. Hence, one application of the Propagation Algorithm is sufficient for verifying (1).

For (2), we must determine if r_i's action A_i can change the effect of r_j's action A_j. We do this by transforming action A_j into a condition C_j such that if the result of condition C_j cannot be affected by the execution of A_i, then A_i cannot change the effect of action A_j. We then apply the Propagation Algorithm to analyze A_i and C_j: if the algorithm produces $\emptyset$, then A_i cannot change the effect of A_j; if the algorithm produces one or more of *insert*, *delete*, or *update*, then A_i may change the effect of A_j.

Consider how condition C_j is derived from action A_j. If A_j is an insert operation, then $A_j = E_{ins}$ is a condition describing the inserted data, hence we let $C_j = E_{ins}$. Similarly, if A_j is a delete operation, then $A_j = E_{del}$ is a condition describing the deleted data, and we let $C_j = E_{del}$. Suppose A_j is an update operation on attribute A, defined by $E_{upd} = \mathcal{E}[A' = expr]E_c$.[5] We start

[5]The extension to multiple updated attributes is obvious. Note that here we require updates to be specified with explicit use of E_c; this does not limit expressiveness.

with the "selection condition" E_c. C_j is the projection of E_c onto all attributes referenced within E_c together with all attributes referenced in the $\mathcal{E}$ operation (both A and the attributes referenced in *expr*). If any of these attributes can be affected by the execution of A_i, then A_i may change the effect of A_j's update; if not, then A_i cannot change the effect of A_j. By using the projection here, rather than the entire expression E_c, we ignore modifications to attributes that do not affect the evaluation of E_c or the assignment of the new values to the updated attribute.

Finally, we analyze (3) by reversing the roles of r_i and r_j in the analysis of (1) and (2).

5.1 Examples

Consider the rules from Section 2.3. We present two examples where we apply our analysis techniques to determine that a pair of rules are commutative (and hence the set of these two rules is confluent). In both of these examples, the technique in [AWH92] is unable to determine that these rules commute.

Example 5.1: Consider rule `bad-account` (Example 2.1) and rule `SF-bonus` (Example 2.3), that here will be called r_1 and r_2 respectively. Both rules reference attribute `rate` and both update this attribute. Hence, intuitively (and according to the method in [AWH92]), the two rules may not commute. We first analyze the effect of r_1's action on r_2. Since r_2's condition does not reference the relation updated by r_1, r_1's action trivially cannot affect r_2's condition. We use the Propagation Algorithm to analyze the effect of r_1's action on the condition corresponding to r_2's action: $\pi_{\texttt{balance,rate,name,city}} E_c$. The input to the algorithm is:

$$C = \pi_{\texttt{balance,rate,name,city}}(\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}}(\texttt{ACCOUNT} \ltimes_{name} (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})))$$
$$A = E_{upd} = \mathcal{E}[\texttt{rate}' = 0](\sigma_{\texttt{balance<500} \wedge \texttt{rate>0}}\texttt{ACCOUNT})$$

The propagation of E_{upd} through the semijoin operation in C yields:

$$E'_{upd} = E_{upd} \ltimes_{name} (\sigma_{\texttt{city='SF'}}\texttt{CUSTOMER})$$

The propagation of E'_{upd} through the selection operation in C yields:

$$E''_{ins} = \rho_{new}((\sigma_{\texttt{balance>5000} \wedge \texttt{rate'<3}} E'_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}} E'_{upd}))$$
$$E''_{del} = \rho_{old}((\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}} E'_{upd}) \ltimes_{\nexists} (\sigma_{\texttt{balance>5000} \wedge \texttt{rate'<3}} E'_{upd}))$$
$$E''_{upd} = (\sigma_{\texttt{balance>5000} \wedge \texttt{rate'<3}} E_{upd}) \bowtie (\sigma_{\texttt{balance>5000} \wedge \texttt{rate<3}} E_{upd})$$

In all three expressions, predicates `balance > 5000` and `balance < 500` (the latter from E_{upd}) are contradictory, so the expressions are unsatisfiable. Hence, the Propagation Algorithm produces no actions and we conclude that executing r_1's action cannot change the effect of r_2's action.

A similar analysis reveals that r_2's action cannot affect r_1's action, and we have already shown in Example 3.1 that r_2's action cannot affect r_1's condition. Hence, we conclude that rules r_1 and r_2 commute.

Example 5.2: Consider rule `add-to-bad` (Example 2.4) and rule `delete-from-bad` (Example 2.5) that here will be called r_1 and r_2 respectively. We have already shown in Example 4.2 that r_1's action cannot affect r_2's condition. An analogous analysis shows that r_1's action cannot affect the condition corresponding to r_2's action, i.e. E_{del}. Consider the effect of rule r_2 on rule r_1. We first apply the Propagation Algorithm to r_2's action and r_1's condition. The input to the algorithm is:

$$C = \pi_{\texttt{num,balance}}((\sigma_{\texttt{balance<500}}\texttt{ACCOUNT}) \ltimes_{\nexists num} \texttt{LOW-ACC})$$
$$A = E_{del} = \texttt{LOW-ACC} \ltimes_{num} (\sigma_{\texttt{balance}\geq\texttt{500}}\texttt{ACCOUNT})$$

where *num* is an abbreviation for `LOW-ACC.num = ACCOUNT.num`. The propagation of E_{del} through the $\ltimes_{\nexists}$ operation in C yields:

$$E'_{ins} \quad = \quad \textit{low-bal} \ltimes_{num} (\texttt{LOW-ACC} \ltimes_{num} \textit{high-bal})$$

where *low-bal* is an abbreviation for $\sigma_{\texttt{balance<500}}\texttt{ACCOUNT}$ and *high-bal* is an abbreviation for $\sigma_{\texttt{balance}\geq\texttt{500}}\texttt{ACCOUNT}$. This expression is not satisfiable, as it requires a tuple with a given `num` value to satisfy both predicates $\texttt{balance} < 500$ and $\texttt{balance} \geq 500$. Hence, r_2's action cannot affect r_1's condition. An analogous analysis shows that r_2's action cannot affect the condition corresponding to r_1's action, i.e. E_{ins}. Hence, we conclude that rules r_1 and r_2 commute.

6 Conclusions and Future Work

We have defined a representation of Condition-Action expert database rules based on an extended relational algebra, and we have described a generally applicable algorithm for analyzing the interactions between one rule's condition (a query) and another rule's action (a modification). We have shown how this algorithm is applied to check termination and confluence for sets of rules. Our technique improves considerably upon previous methods, because our formal approach allows us to exploit the semantics of conditions and actions to analyze the interaction between rules. Note that the methods we describe also are applicable to rule languages that "pass data" from the condition to the action (e.g. [GP91,Han92]), since our algorithm detects the actual modifications to rule conditions (inserts, deletes, and updates), not simply the transition between true and false. As in [AWH92], our analysis techniques identify the responsible rules when termination or confluence is not guaranteed; hence, our techniques can be used as the kernel of an interactive development tool that helps rule definers develop sets of rules that are guaranteed to have the desired properties.

We plan to extend our rule model and analysis techniques to incorporate additional features of expert database rules:

- **Rule priorities and conflict resolution.** Priorities restrict the possible execution sequences of rules, making analysis more complex but perhaps more precise. Coupling our accurate analysis of rule interactions with the priority-based methods in [AWH92] should immediately produce a quite powerful analysis method for prioritized rules.

- **Different semantics for rule condition evaluation.** In some database rule languages, rule conditions may be evaluated over the entire database, as opposed to considering only "new" data as we have done here. This interpretation yields a different notion of rule activation, since a rule condition remains true unless execution of some rule action renders it false.

- **Events.** We can handle Event-Condition-Action rules that have a semantics similar to our Condition-Action rules, e.g. the event-based rules of Ariel [Han92], with minor modifications to our techniques. (In fact, it is our feeling that event-based rules often are programmed this way in practice, e.g. [CW90].) However, general Event-Condition-Action rules, especially those in which the condition is evaluated over the entire database, will require a redefinition of rule activation (as discussed in the previous point), along with corresponding modifications to our method.

We also hope to use our algebraic rule model and Propagation Algorithm as the basis for compile-time and run-time optimizations to rule processing.

Acknowledgements

We are grateful to members of the Stanford Database Group, especially Ashish Gupta and Jeff Ullman, for lively and useful discussions about this work, and to Stefano Ceri for providing the technical impetus and enabling the collaboration.

References

[AWH92] A. Aiken, J. Widom, and J.M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[Bar94] E. Baralis. *An Algebraic Approach to the Analysis and Optimization of Active Database Rules.* PhD thesis, Politecnico di Torino, Torino, Italy, February 1994.

[BCW93] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In *Proceedings of the First International Workshop on Rules in Database Systems*, Edinburgh, Scotland, August 1993.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[BM93] D.A. Brant and D.P. Miranker. Index support for rule activation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 42–48, Washington, D.C., May 1993.

[BW93] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. Technical Report Stan-CS-93-1495, Computer Science Department, Stanford University, November 1993.

[CCRL+90] S. Ceri, S. Crespi-Reghizzi, L. Lamperti, L. Lavazza, and R. Zicari. Algres: An advanced database for complex applications. *IEEE Software*, June 1990.

[CG85] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, April 1985.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[DE89] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: A production language for relational databases. In L. Kerschberg, editor, *Expert Database Systems—Proceedings from the Second International Conference*, pages 333–351. Benjamin/Cummings, Redwood City, California, 1989.

[DOS+92] H.M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, and S. Da Silva. Incremental database rule processing in PARADISER. *Journal of Intelligent Information Systems*, 1992.

[Elk90] C. Elkan. Independence of logic database queries and updates. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–160, March 1990.

[GP91] D.N. Gordin and A.J. Pasik. Set-oriented constructs: From Rete rule bases to database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 60–67, Denver, Colorado, May 1991.

[Han92] E.N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, San Diego, California, June 1992.

[HH91] J.M. Hellerstein and M. Hsu. Determinism in partially ordered production systems. IBM Research Report RJ 8009, IBM Almaden Research Center, San Jose, California, March 1991.

[HW93] E.N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–727, 1982.

[KU94] A.P. Karadimce and S.D. Urban. Conditional term rewriting as a formal basis for analysis of active database rules. In *Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS '94)*, Houston, Texas, February 1994.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 171–181, Dublin, Ireland, August 1993.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, September 1991.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 455–464, Amsterdam, The Netherlands, August 1989.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational DBMS. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 315–326, Vancouver, British Columbia, August 1992.

[SLR88] T. Sellis, C.-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 404–412, Chicago, Illinois, June 1988.

[Tzv88] A. Tzvieli. On the coupling of a production system shell and a DBMS. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 291–309, Jerusalem, Israel, June 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.

[vdVS93] L. van der Voort and A. Siebes. Termination and confluence of rule execution. In *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, November 1993.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology—EDBT '90, Lecture Notes in Computer Science 416*, pages 407–421. Springer-Verlag, Berlin, March 1990.