In: *Information Systems*, 19(6):467–490, 1994

Deriving Incremental Production Rules for Deductive Data*

Stefano Ceri
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
I-20133 Milano, Italy
ceri@ipmel2.elet.polimi.it

Jennifer Widom
Department of Computer Science
Stanford University
Stanford, CA 93405-2140
USA
widom@cs.stanford.edu

Abstract

We show that the production rule mechanism provided by active database systems can be used to quickly and easily implement the logic rule interface of deductive database systems. Deductive rules specify derived relations using Datalog with built-in predicates and stratified negation; the deductive rules are compiled automatically into production rules. We present a materialized approach, in which the derived relations are stored in the database and the production rules automatically and incrementally propagate base relation changes to the derived relations. We also present a non-materialized approach, in which the production rules compute the derived relations on demand.

1 Introduction

A considerable amount of research has focused on adding rules to database systems. This work is divisible into two areas: *deductive* database systems and *active* database systems. In deductive database systems, logic programming style rules are used to provide a more powerful user interface than that provided by most database query languages [CGT90,Min88,Ull89]. In active database systems, production style (forward-chaining) rules are used to provide automatic execution of database operations in response to certain events and/or conditions [DHW94,HW93].

With the rapid emergence of production rule capabilities in research prototypes, in a number of commercial database systems, and in the upcoming SQL3 standard, we believe that active databases will become widely available in practice. However, we also believe that production rules in database systems should be treated as a low-level mechanism, used to implement higher-level functionality [Cer92]. In this paper we show that production rules can be used to easily implement the high-level interface provided by deductive databases. Hence, our work will make it possible to quickly support deductive capabilities in widely available systems.

In deductive databases, there is a distinction between *base* (or *extensional*) data and *derived* (or *intensional*) data. In this paper, we assume all data is stored in *relations*. Base relations are created and manipulated by users in the usual way. Derived relations are defined by deductive rules specified in a language analogous to *Datalog* [Ull89], which in our framework includes built-in predicates and stratified negation. Derived relations can be *materialized*, in which case they are

*This work was partially performed while both authors were at the IBM Almaden Research Center, San Jose, CA. In Milano, this work was partially supported by Esprit Project P6333, "Idea". At Stanford, this work was partially supported by the Reid and Polly Anderson Faculty Scholar Fund and by an equipment grant from IBM Corporation.

stored in the database and kept consistent with the base relations over which they are defined, or they can be *non-materialized*, in which case they are not stored in the database but are computed from the base relations as needed.

In our framework, each deductive rule is compiled automatically into a set of production rules. Our main result concerns materialized derived relations. In this context, the generated production rules are triggered by changes to base relations that may affect the value of derived relations, and the productions rules automatically modify the derived relations accordingly. The modifications are performed *incrementally*, thus exploiting available information about base relation changes and avoiding complete recomputation of derived relations. We also adapt our approach to non-materialized derived relations. In this context, the generated production rules are triggered by queries referencing derived relations, and the production rules automatically compute the necessary derived relations. Known optimization techniques for the computation, such as *magic sets* [BR86], can be incorporated into the generated production rules. In both the materialized and the non-materialized approach, the generated production rules encode a specific strategy for computing derived relations; however, our framework can easily be adapted for different strategies.

There is a strong similarity between the strategy used by the generated production rules and *semi-naive evaluation* [CGT90,Ull89]. Semi-naive evaluation is an incremental method that relies on *deltas*—changes between a previous database state and the current state. Most database production rule languages provide a mechanism for accessing deltas directly and efficiently [DHW94]. Hence, one of the contributions of our work is to provide an automatic method for exploiting the access to deltas provided by production rules, eliminating the need to "hard code" this access for deductive rule processing.

1.1 Related Work

Although there are substantial bodies of work in both active and deductive databases, only limited research to date has focused on connecting the two approaches. Some work has shown how deductive databases can be extended to support active behavior, e.g [BJ93, CCCR$^+$90, HD93b, SKdM92,Tan91]; this relates to the converse of the problem considered here. In the *RDL1* system, a forward-chaining production style of rule processing is used to implement a deductive database language [KdMS90]. The goal of RDL1 is an efficient deductive database system. Hence, although the underlying implementation is based on rule triggering, the rule language is deductive. In our case, we illustrate how deductive rules can be supported using the existing facilities of an active database system. This paper considerably improves upon and extends our own initial work in this area [Wid91].

In the approach we take to materialized derived relations, production rules propagate base relation changes to derived relations incrementally. Other research that has considered the problem of incrementally maintaining derived data includes [AP87, DT92, GMS93, HD93a, Kuc91, NY83, UO92,WDSY91]. Similar approaches have been applied to integrity constraint maintenance, e.g. [Nic82,UKN92,CFPT94,CW90], where the incremental nature of the computation results from the assumption that the previous database state satisfies the constraints.

Pioneering work in incremental maintenance of deductive data [NY83] proposes an approach to materialized derived relations in which a counter is maintained for each derived tuple, encoding its number of derivations. Algorithms are given that use the counters to efficiently maintain the derived relations when base relations are modified. A similar approach based on derivation counting is described in [GMS93]. In [DT92,Kuc91,WDSY91], the actual derivations of tuples are encoded in auxiliary information stored with the derived relations. Changes to base relations are first considered with respect to the auxiliary information, then propagated to the derived relations. In contrast to all of these approaches, our method does not require any auxiliary information—derived relations are maintained incrementally using only the behavior provided by production rules. The absence of auxiliary information enables us to maintain derived relations using active database rules only, without any modifications to the underlying database system.

Methods in [GMS93, HD93a] also do not require auxiliary information; both papers propose algorithms that effectively "undo" and then "redo" derivations, which is similar to the effect of our production rules for materialized derived relations without unique derivations. In [UO92], special *internal events* are defined that describe transitions on derived relations, with rules for incremental evaluation of these events. In [AP87], a technique is presented based on *belief revisions*: procedures are defined for deducing the effect of insertions or deletions of data or rules on a given database that represents the current *belief*. Although the context and procedures in [AP87] are very different from our approach, there are some similarities (such as the treatment of stratification).

In [CW91], we describe a method for deriving production rules that incrementally maintain materialized views, where views are specified using a subset of SQL. Since there is an overlap between SQL views and derived relations specified using Datalog, there also is an overlap between the methods in [CW91] and the methods given here. In particular, views that are defined to be *safe* in [CW91] correspond to derived relations with *unique derivations* in this paper, so simplifications introduced for safe views in [CW91] carry over; see Section 6. In this paper, however, we handle derived relations that are defined recursively and may be defined through multiple deductive rules. Furthermore, we give a fully incremental approach, even for derived relations with multiple derivations, thereby generalizing and improving on the results in [CW91] considerably. Finally, note that much of the mechanism in [CW91] involves handling the complexities and limitations of views defined using SQL; these complexities and limitations are not an issue here.

1.2 Outline of Paper

Section 2 describes the initial language we use for deductive rules (excluding negation) and presents a running example for the paper. Section 3 describes the language we use for production rules—the rule language of *Starburst*.[1] Section 4 contains the first core results of the paper: a method for translating deductive rules in our initial language into production rules that incrementally

[1] We use the Starburst rule language since we have developed and can experiment with it, but the same approach can be applied using other database production rule languages.

maintain materialized derived relations, and a proof that the method is correct. Section 5 extends these results for deductive rules with stratified negation. Section 6 explains how the generated production rules can be simplified when derived relations are known to have unique derivations. Section 7 describes our approach for non-materialized derived relations. Section 8 concludes and describes future work.

2 Deductive Rules

The initial language we use for deductive rules is based on Datalog with built-in predicates but without negation. (We extend our results to stratified negation in Section 5.) The description given here is brief and somewhat informal; for thorough treatments of deductive database rule languages see, e.g., [BR86,CGT90,Ull89].

Let the database consist of a set of relations, some designated as base relations and others designated as derived relations.[2] Users may perform arbitrary retrieval or modification operations on base relations (e.g. SQL **select**, **insert**, **delete**, **update**).[3] Users may perform only retrieval operations on derived relations—data in derived relations is specified exclusively by deductive rules. The general form of a deductive rule is:

$$\begin{aligned} \textit{rule-name}:\ D(E_1,\ldots,E_i) \text{ :- } & R_1(X_1^1,\ldots,X_{j_1}^1),\ \ \ldots,\ \ R_n(X_1^n,\ldots,X_{j_n}^n), \\ & P_1(E_1^1,\ldots,E_{k_1}^1),\ \ \ldots,\ \ P_m(E_1^m,\ldots,E_{k_m}^m) \end{aligned}$$

with the following requirements:

- D names a derived relation.
- $R_1,\ldots,R_n$ name base or derived relations.
- $P_1,\ldots,P_m$ are built-in predicates. Any predicate evaluable by the database query language is allowed. (Since we will be considering the Starburst database system [H$^+$90], these are the standard SQL predicates along with user-defined predicates.)
- All X's on the right-hand-side (RHS) of the rule are either constants (specified by value) or variable names.
- All E's on the left-hand-side (LHS) and RHS of the rule are either constants, variable names that also appear at least once on the RHS as an X, or expressions over constants and such variables. Any expression evaluable by the database query language is allowed. (Again, in Starburst these are the standard SQL expressions along with user-defined expressions.)
- The number and type of arguments to D and $R_1,\ldots,R_n$ match the schema of the relations.

[2]Although the only derived data we consider in this paper is relations, a similar approach could certainly be used to support, e.g., derived values stored as relational attributes.

[3]For simplicity in this paper we treat update operations as deletes followed by inserts (see Section 4), but an extension to directly handle updates is straightforward.

Recursion is allowed, and relations may appear multiple times on the RHS.

Our rules have the standard deductive interpretation [CGT90, Ull89]: Consider every set of tuples, one each from relations $R_1, \ldots, R_n$, satisfying the following conditions:

1. All RHS X's that are constants match the values in the corresponding tuple fields.

2. If all RHS X's that are variables are assigned values from the corresponding tuple fields then each variable is assigned a unique value (i.e. common variables hold the same value) and predicates $P_1, \ldots, P_m$ are satisfied.

For each set of tuples satisfying these conditions, the corresponding tuple defined by $E_1, \ldots, E_i$ using the variable assignments from condition 2 is in relation D.

Note that since our deductive rules include predicates and expressions, infinite derived relations are possible. We do not exclude infinite derived relations from our framework. In the case of materialized derived relations, and sometimes in the case of non-materialized derived relations, the effect of infinite derived relations is non-terminating execution of the generated production rules (which in most systems produces an error). Also note that, although we allow users to create duplicate tuples in base relations, derived relations do not have duplicates.

For convenience, we abbreviate the generic deductive rule specified above as:

$$\textit{rule-name}: \; D(\bar{E}) \;\text{:-}\; R_1(\bar{X}_1),\ \ldots,\ R_n(\bar{X}_n),\ P(\bar{E}')$$

so that an argument list $A_1, \ldots, A_i$ (say) is denoted by $\bar{A}$, and predicates $P_1, \ldots, P_m$ are conjoined into a single predicate P. (Our language for predicates includes logical **and**.)

2.1 Running Example

We introduce a simple example to be used throughout the paper. We consider two base relations:

```
Station(city, state)
Train(city1, city2)
```

Station contains cities with train stations; for brevity we assume that attribute *city* is a key for this relation. *Train* contains pairs of cities such that there is direct train service from the first city to the second; *city1* and *city2* together form a key for this relation. We also consider two derived relations:

```
Route(city1, city2)
Reach-Cal(city)
```

Route contains pairs of cities such that there is a train route from the first city to the second city. (That is, *Route* is the transitive closure of *Train*.) *Reach-Cal* contains those cities from which it is possible to reach a city in California by train. The following four deductive rules define the derived relations:

```
rt1: Route(C1,C2) :- Train(C1,C2)
rt2: Route(C1,C2) :- Route(C1,C3), Route(C3,C2)
rc1: Reach-Cal(C) :- Station(C,S), S = "California"
rc2: Reach-Cal(C) :- Route(C,C'), Reach-Cal(C')
```

Although this example does not include all of the features in our deductive rule language (excluding, e.g., complex expressions), it is sufficiently comprehensive to illustrate our approach.

3 Production Rules

This section introduces the *Starburst Rule System*, an active database extension to the Starburst prototype DBMS [H$^+$90] supporting the language we use for production rules. Although developed in the context of a specific research project, the Starburst rule language is representative of SQL-like production rule languages. In addition, the Starburst rule language has been formally defined and implemented [WCL91,WF90], unlike e.g. the proposal for triggers in SQL3, permitting us to both prove the correctness of our approach and to experiment with our methods on a running system. Our methods certainly can be adapted to other database production rule languages, and ultimately to SQL3.

We first introduce a convenient shorthand notation for the subset of the Starburst rule language used by our rule generation methods. We then describe the actual language supported by Starburst and explain how our shorthand notation maps to this language. We also describe the Starburst rule processing model, and we provide a few examples. (More examples appear throughout the paper.) Further details on the Starburst Rule System can be found in [WCL91,WF90].

3.1 Notation

Starburst production rules rely on the notion of *transitions*—database state changes resulting from execution of a sequence of data manipulation operations. Rules are *triggered* by the effect of transitions. Triggered rule *actions* perform additional data manipulation operations, creating additional transitions. Further details of rule processing semantics are deferred until Section 3.3.

The shorthand notation we use for expressing the production rules generated by our methods is as follows:

$$\textit{rule-name}:\ \{\mathbf{ins}\,|\,\mathbf{del}\}\ D(\bar{E}) \leftarrow [\mathbf{old}]\ R_1(\bar{X}_1),\ \ldots,\ \{\mathbf{ins}\,|\,\mathbf{del}\}\ R_i(\bar{X}_i),\ \ldots,\ [\mathbf{old}]\ R_n(\bar{X}_n),\ P(\bar{E}')$$

Names and symbols satisfy the same requirements that were introduced in Section 2 for deductive rules. In addition, D is tagged with either **ins** or **del**, exactly one R_i is tagged with either **ins** or **del**, and each R_j other than R_i may be tagged with **old**.

The meaning of this rule is the following. First suppose that the rule does not contain any **old**'s, and that both D and R_i are tagged with **ins**. The **ins** tag on relation R_i indicates that the rule is triggered by any transition in which there are insertions into relation R_i. Once the rule is triggered, the rule's action retrieves every set of tuples, one each from relations $R_1, \ldots, R_n$ except using only inserted tuples from R_i, such that:

1. All RHS X's in the rule (as specified above) that are constants match the values in the corresponding tuple fields.

2. If all RHS X's that are variables are assigned values from the corresponding tuple fields then each variable is assigned a unique value and predicates $P_1, \ldots, P_m$ are satisfied.

For each set of tuples retrieved, the rule's action inserts the tuple defined by $\bar{E}$ using the variable assignments from condition 2 into relation D, if the tuple is not in D already. (Informally, this corresponds to one "application" of a deductive rule according to the semantics given in Section 2.)

If D is tagged with **del** instead of **ins**, then the tuple defined by $\bar{E}$ is deleted from relation D rather than inserted. If R_i is tagged with **del** instead of **ins**, then the rule is triggered by deletions instead of insertions, and deleted tuples from R_i are used instead of inserted tuples. Finally, if a relation R_j is tagged with **old**, then instead of using R_j's current value, the rule uses R_j's value preceding the rule's triggering transition.

3.2 Rules in Starburst

In the Starburst Rule System, the syntax for creating a rule is:

> **create rule** *name* **on** *relation*
> **when** *triggering operations*
> [**if** *condition*]
> **then** *action*
> [**precedes** *rule-list*]
> [**follows** *rule-list*]

The *triggering operations* are one or more of **inserted**, **deleted**, and **updated**. The optional *condition* is an arbitrary SQL predicate. (Our methods do not use rule conditions.) The *action* is an arbitrary sequence of SQL data manipulation operations. The optional **precedes** and **follows** clauses are used to partially order the set of rules: If a rule r_1 specifies a rule r_2 in its **precedes** list, or if r_2 specifies r_1 in its **follows** list, then r_1 has priority over r_2. When no (direct or transitive) ordering is specified between two rules, their relative priority is arbitrary. Although we have not included rule priority specifications in our shorthand notation, our method does exploit rule priorities, as will be seen below.

Rule conditions and actions may refer to the current state of the database through top-level or nested SQL **select** operations. In addition, rule conditions and actions may refer to *transition tables*, which are logical relations reflecting the changes that have occurred during a rule's triggering transition. Rules consider only the *net effect* of transitions, as defined in [WF90]. At the end of a given transition, transition table **inserted** in a rule refers to those tuples of the rule's relation that were inserted by the transition. Transition tables **deleted**, **new-updated**, and **old-updated** are similar. Note that transition tables correspond exactly to the notion of *deltas* discussed in Section 1.

Although Starburst rules have a fairly complex, SQL-like syntax, the simpler notation introduced for our purposes in Section 3.1 maps directly into Starburst rules. For each production rule defined using the shorthand notation, we automatically produce a Starburst rule with the following structure:

> **create rule** *rule-name* **on** R_i
> **when** { **inserted** | **deleted** }
> **then** { **insert into** D *Ins-Select-Expression* |
> **delete from** D **where** $\langle D.attributes \rangle$ **in** (*Del-Select-Expression*) }

The rule's triggering operation depends on whether R_i is tagged with **ins** or **del**, and the operation performed by the rule's action depends on whether D is tagged with **ins** or **del**. The *Del-Select-Expression* is an SQL **select** operation that produces all tuples defined by $\bar{E}$ according to conditions 1 and 2 described above. (Hence, in the **select** operation, transition table **inserted** or **deleted** is used in place of R_i, and some relations may be prefaced by **old**.) The *Ins-Select-Expression* is identical to the *Del-Select-Expression*, except **distinct** and a **not in** clause are added so that duplicate tuples are not inserted. Note that Starburst rules currently do not support direct access to **old** relation values, but these values can be computed using transition tables; see [CW91].

We omit the details of deriving the *Select-Expressions* here. Methods for translating deductive rule RHS's into SQL **select** operations are given in [CGT90,Ull89], and a detailed modification of these methods appropriate for use here is presented in [Wid91]. Examples in Section 3.4 should convince the reader that the translation, although somewhat tedious to specify, is straightforward. Many additional examples of the translation appear in the Appendix.

Finally, note that our SQL operations use multi-attribute **in** and **not in**, which is available in some but not all SQL implementations; equivalent expressions in standard SQL can be substituted.

3.3 Rule Processing

In Starburst, rules are processed at *rule processing points*. There is an automatic rule processing point at the end of each transaction, and there may be additional user-specified processing points within a transaction. We describe rule execution at an arbitrary processing point. Since rule conditions are not used in this paper, we omit them them from our description of rule processing.

The state change resulting from the user-generated database operations executed since the last rule processing point (or start of the transaction) creates the first relevant transition, and some set of rules are triggered by this transition. A triggered rule r is chosen from this set for *execution*. Rule r is chosen so that no other triggered rule has priority over r, and r's action is executed. After execution of r's action, all rules other than r are triggered if a triggering operation occurred in the composite transition created by the initial transition and subsequent execution of r's action. Rule r is triggered again only if a triggering operation occurred in the transition created by its action. From the new set of triggered rules, a rule r' is chosen for execution so that no other triggered rule has priority over r', and rule processing continues in this fashion.

At an arbitrary point in rule processing, a given rule is triggered if a triggering operation occurred in the (composite) transition since the last time it was executed; if it has not yet been executed, it is triggered if a triggering operation occurred in the transition since the start of the transaction. The effect of this semantics is that every rule considers every database modification exactly once. The values of transition tables in rule conditions and actions always reflect the rule's triggering transition. Rule processing terminates when there are no more triggered rules.

3.4 Examples

Our examples use the relations introduced in Section 2.1. Consider the following production rule expressed in our shorthand notation:

```
rc1-ins: ins Reach-Cal(C) <- ins Station(C,S), S = "California"
```

This rule is triggered by insertions into relation *Station*. Its action inserts into relation *Reach-Cal* all of the cities inserted into *Station* whose *state* field is "California". The corresponding production rule in Starburst notation is:

```
create rule rc1-ins on Station
when inserted
then insert into Reach-Cal
        select distinct city from inserted
        where state = "California"
        and city not in (select * from Reach-Cal)
```

As a second example, consider the following production rule expressed in our notation:

```
rt2-del: del Route(C1,C2) <- del Route(C1,C3), old Route(C3,C2)
```

This rule is triggered by deletions from relation *Route*. Its action further deletes from *Route* any tuple whose first attribute corresponds to a tuple t_1 deleted from *Route*, whose second attribute corresponds to a tuple t_2 in the pre-transition value of *Route*, and such that t_1's second attribute matches t_2's first attribute. The corresponding production rule in Starburst notation is:

```
create rule rt2-del on Route
when deleted
then delete from Route where <city1,city2> in
       (select deleted.city1, Route.city2
        from deleted, old Route
        where deleted.city2 = Route.city1)
```

Not surprisingly, we will see in Section 4.2 that both of these examples correspond to production rules generated by our framework from the deductive rules given in Section 2.1.

4 Maintaining Derived Relations

We first consider the materialized approach, in which all derived relations are stored in the database. From each deductive rule, a number of production rules are generated. Although here we use our shorthand notation to describe the production rules, in practice they can be generated directly in

Starburst notation. We assume initially that all derived relations and deductive rules are defined when the database is created, i.e. when the relevant base relations are empty. As users create and modify data in the base relations, these modifications are propagated to the derived relations. In Section 7.2 we explain how derived relations and deductive rules can be added when other relations are non-empty.

Consistency of the derived relations with the base relations is maintained entirely by production rules. The derived relations are consistent at the start of each transaction, and the production rules ensure that they are consistent at the end of each transaction by propagating that transaction's base relation modifications to the relevant derived relations. (Recall from Section 3.3 that production rules are processed automatically at the end of each transaction.) If consistency is desired at points within a transaction, commands can be issued to invoke rule processing at these points.[4]

For clarity and simplicity in the remainder of the paper, we assume that tuples are only inserted into and deleted from base relations. Users may perform updates, but these are treated as deletes followed by inserts. Extending our approach to handle updates directly is quite straightforward; see Section 8.

4.1 Generating Production Rules

Consider the following generic deductive rule:

$$r\colon\ D(\bar{E}) \text{ :- } R_1(\bar{X}_1),\ \ldots,\ R_n(\bar{X}_n),\ P(\bar{E}')$$

From this rule, the following $2n+1$ production rules are generated:

1. *Deletion Rules* – one rule for each i, $1 \le i \le n$

   r-del-i: $\textbf{del}\ D(\bar{E}) \leftarrow \textbf{old}\ R_1(\bar{X}_1),\ \ldots,\ \textbf{del}\ R_i(\bar{X}_i),\ \ldots,\ \textbf{old}\ R_n(\bar{X}_n),\ P(\bar{E}')$

2. *Re-insert Rule*[5]

   r-ri: $\textbf{ins}\ D(\bar{E}) \leftarrow \textbf{del}\ D(\bar{E}),\ R_1(\bar{X}_1),\ \ldots,\ R_n(\bar{X}_n),\ P(\bar{E}')$

3. *Insertion Rules* – one rule for each i, $1 \le i \le n$

   r-ins-i: $\textbf{ins}\ D(\bar{E}) \leftarrow R_1(\bar{X}_1),\ \ldots,\ \textbf{ins}\ R_i(\bar{X}_i),\ \ldots,\ R_n(\bar{X}_n),\ P(\bar{E}')$

The rules are partially ordered (using Starburst's **precedes** and **follows** clauses) so that deletion rules have priority over the re-insert rule which has priority over insertion rules.

To explain these rules, and to prove their correctness in Section 4.3, we digress briefly to describe the notion of *derivations*. Each tuple t in a derived relation has one or more derivations. A derivation can be represented by a tree, where each node in the tree is a tuple: leaf nodes are

[4] Starburst also supports commands to invoke rule processing with only a subset of the production rules [Wid92]. This feature can be used within a transaction to propagate base relation modifications to certain derived relations while leaving other derived relations for end-of-transaction rule processing.

[5] Actually, $D(\bar{E})$ in the RHS of this rule may violate our syntactic requirements if $\bar{E}$ contains complex expressions. However, any expression E in $\bar{E}$ can equivalently be replaced by a new variable V with $V = E$ conjoined to predicate P.

base relation tuples, interior nodes are derived relation tuples, and the root node is derived tuple t. For each non-leaf node, the parent tuple can be deduced from its child tuples by one application of one deductive rule (recall the interpretation of deductive rules given in Section 2). Hence, each derivation tree is a complete description of how a tuple in a derived relation can be produced from tuples in base relations. As a simple illustration, consider the following deductive rule:

```
r: D(A,C) :- R(A,B), R(B,C)
```

and suppose base relation R contains the tuples `(a,b)`, `(b,c)`, and `(c,d)`. Then one derivation tree for tuple `(a,d)` in derived relation D is:

```
              D(a,d)
              /    \
          D(a,c)    R(c,d)
          /    \
     R(a,b)    R(b,c)
```

Now consider the set of production rules given above. The deletion rules execute first; they delete tuples from D based on deletions from RHS relations. Intuitively, when a tuple is deleted from a RHS relation, any derivation of a tuple in D based on rule r and the deleted tuple is no longer valid. Hence, each time a deletion rule executes, it deletes all tuples from D whose derivation relies on one application of rule r using a tuple deleted from R_i. When a deletion rule deletes tuples from D, it will trigger any additional deletion rules (including itself) for which **del** D appears on the RHS. Note that at the time the deletion rules execute, the tuples stored in D are consistent with the previous (i.e. pre-transition) state of the RHS relations, not the current state. Hence, to identify the correct values for the tuples to be deleted from D, it is necessary to use pre-transition values for the RHS relations, as specified by our use of **old**.

Tuples in derived relations may have multiple derivations. Consequently, the deletion rules may delete a tuple because one of its derivations is no longer valid, but that tuple may have other derivations that are valid. Hence, after execution of the deletion rules, the re-insert rule inserts into D tuples that were deleted from D by the deletion rules, but that should remain in D because they have other valid derivations. This rule executes only once, so it only re-inserts deleted tuples with "one-step" derivations. However, any tuples inserted by this rule can trigger and be considered by the insertion rules, which we explain next.

The insertion rules execute last. They insert tuples into D based on insertions into RHS relations. Intuitively, when a tuple is inserted into a RHS relation, it may create new derivations for new tuples in D. Hence, each time an insertion rule executes, it inserts into D tuples (not already in D) whose derivation relies on one application of rule r using a tuple inserted into R_i. When an insertion rule inserts tuples into D, it will trigger any additional insertion rules (including itself) for which **ins** D appears on the RHS.

We have described the production rules generated from a single deductive rule. Now suppose there are multiple deductive rules. Multiple deductive rules may share LHS relations, may share

RHS relations, and may be mutually recursive; these features are handled automatically by our rule generation mechanism. Production rules are generated as described above for each deductive rule. The entire set of rules is partially ordered so that all deletion rules have highest priority, all re-insert rules have middle priority, and all insertion rules have lowest priority. (There is no need to order rules within each group.) Hence, rule processing consists of three "phases": in the first phase tuples are deleted from derived relations, in the second phase deleted tuples are re-inserted into derived relations, and in the third phase additional tuples are inserted into derived relations.[6]

The generated production rules are incremental—each rule modifies data in derived relations based on modifications to data in base or derived relations. This is reflected in the fact that the RHS of each generated rule (in our notation) has one relation tagged with either **ins** or **del**. In practice, the amount of modified data is expected to be substantially less than the amount of existing data, making incremental rules very efficient. If, in fact, the amount of modified data is large, our incremental rules will effectively delete and then recompute all derived data.[7]

4.2 Example

Consider the four deductive rules introduced in Section 2.1:

```
rt1: Route(C1,C2) :- Train(C1,C2)
rt2: Route(C1,C2) :- Route(C1,C3), Route(C3,C2)
rc1: Reach-Cal(C) :- Station(C,S), S = "California"
rc2: Reach-Cal(C) :- Route(C,C'), Reach-Cal(C')
```

Using our notation, the following production rules are generated:

```
rt1-del-1: del Route(C1,C2) <- del Train(C1,C2)
rt2-del-1: del Route(C1,C2) <- del Route(C1,C3), old Route(C3,C2)
rt2-del-2: del Route(C1,C2) <- old Route(C1,C3), del Route(C3,C2)
rc1-del-1: del Reach-Cal(C) <- del Station(C,S), S = "California"
rc2-del-1: del Reach-Cal(C) <- del Route(C,C'), old Reach-Cal(C')
rc2-del-2: del Reach-Cal(C) <- old Route(C,C'), del Reach-Cal(C')

rt1-ri: ins Route(C1,C2) <- del Route(C1,C2), Train(C1,C2)
rt2-ri: ins Route(C1,C2) <- del Route(C1,C2), Route(C1,C3), Route(C3,C2)
rc1-ri: ins Reach-Cal(C) <- del Reach-Cal(C), Station(C,S), S = "California"
rc2-ri: ins Reach-Cal(C) <- del Reach-Cal(C), Route(C,C'), Reach-Cal(C')

rt1-ins-1: ins Route(C1,C2) <- ins Train(C1,C2)
rt2-ins-1: ins Route(C1,C2) <- ins Route(C1,C3), Route(C3,C2)
```

[6] While the general approach should be clear, it is understandable if the reader does not fully grasp all intricacies of production rule behavior and interaction at this point. Readers interested in and/or skeptical about details are encouraged to read the correctness proof in Section 4.3.

[7] This can be achieved more efficiently using our non-materialized approach (see Section 7), which may be more appropriate than the materialized approach if frequent large modifications are expected.

```
rt2-ins-2: ins Route(C1,C2) <- Route(C1,C3), ins Route(C3,C2)
rc1-ins-1: ins Reach-Cal(C) <- ins Station(C,S), S = "California"
rc2-ins-1: ins Reach-Cal(C) <- ins Route(C,C'), Reach-Cal(C')
rc2-ins-2: ins Reach-Cal(C) <- Route(C,C'), ins Reach-Cal(C')
```

The production rules are ordered so that all rules in the first group have priority over all rules in the second group which have priority over all rules in the third group. (By transitivity, all rules in the first group have priority over all rules in the third group.) The corresponding rules in Starburst notation are given in Appendix A.1.

4.3 Correctness

Theorem 4.1 Consider arbitrary derived relations $D_1, \ldots, D_m$ defined over arbitrary base relations $B_1, \ldots, B_n$ by arbitrary deductive rules $r_1, \ldots, r_x$. Let $\rho_1, \ldots, \rho_y$ be the production rules generated from $r_1, \ldots, r_x$ as described in Section 4.1. Suppose $D_1, \ldots, D_m$ are consistent initially with $B_1, \ldots, B_n$, then arbitrary tuples are deleted from and/or inserted into $B_1, \ldots, B_n$, then rule processing is invoked with $\rho_1, \ldots, \rho_y$. When rule processing terminates,[8] $D_1, \ldots, D_m$ are consistent with the new values of $B_1, \ldots, B_n$.

Proof: Let $\bar{B}_0$ denote $B_1, \ldots, B_n$ before the modifications, let $\bar{B}$ denote $B_1, \ldots, B_n$ after the modifications, let $\bar{D}_0$ denote $D_1, \ldots, D_m$ before rule processing, and let $\bar{D}$ denote $D_1, \ldots, D_m$ consistent with $\bar{B}$. By assumption, $\bar{D}_0$ is consistent with $\bar{B}_0$ and $\bar{D}$ is consistent with $\bar{B}$; we must show that rule processing modifies $D_1, \ldots, D_m$ from $\bar{D}_0$ to $\bar{D}$. The proof is based on derivation trees (recall Section 4.1) and proceeds in two steps: in the first step we show that any tuple that is in $\bar{D}_0$ but not in $\bar{D}$ is deleted by rule processing; in the second step we show that any tuple that is in $\bar{D}$ but is not in $\bar{D}_0$, or that is in $\bar{D}$ but is deleted from $\bar{D}_0$ by rule processing, is inserted by rule processing. It is straightforward to see that rule processing cannot insert tuples that are not in $\bar{D}$, so these two steps complete the proof.

[1] *Deletions*: Let t be a tuple in $\bar{D}_0$ but not in $\bar{D}$. Then any derivation tree for t in $\bar{D}_0$ is not valid in $\bar{D}$; consider one such tree T. We prove by induction on the depth of T that t is deleted by rule processing.

Base case – T has depth 1. Let t's children in T be base tuples $b_1, \ldots, b_k$, with t derived from its children by deductive rule r. Since T is not valid in $\bar{D}$, at least one b_i must have been deleted. Production rule r-del-i generated from r is triggered by b_i's deletion and deletes t.

Induction – T has depth $n > 1$. Let t's children in T be base and derived tuples $t_1, \ldots, t_k$, with t derived from its children by deductive rule r. Since T is not valid in $\bar{D}$, at least one of t's children is a deleted base tuple b_i or a derived tuple d_j whose subtree is not valid in $\bar{D}$. If one of t's children is a deleted base tuple b_i, then production rule r-del-i generated from r is triggered by b_i's deletion

[8] Rule processing is guaranteed to terminate if and only if all derived relations consistent with the new base relations are finite.

and deletes t. If one of t's children is a derived tuple d_j whose subtree is not valid in $\bar{D}$, then, by the induction hypothesis, d_j is deleted by rule processing; production rule r-del-j generated from r is triggered by d_j's deletion and deletes t.

[2] *Insertions*: Let t be a tuple in $\bar{D}$ that is either (A) deleted from $\bar{D}_0$ by rule processing or (B) not in $\bar{D}_0$. Consider any derivation tree T for t in $\bar{D}$. We prove by induction on the depth of T that t is inserted by rule processing.

Base case – T has depth 1. Let t's children in T be base tuples $b_1, \ldots, b_k$, with t derived from its children by deductive rule r. In case (A), production rule r-ri generated from r is triggered by t's deletion and re-inserts t. In case (B), since T is not valid in $\bar{D}_0$, at least one b_i must have been inserted. Production rule r-ins-i generated from r is triggered by b_i's insertion and inserts t.

Induction – T has depth $n > 1$. Let t's children in T be base and derived tuples $t_1, \ldots, t_k$, with t derived from its children by deductive rule r. Note that each derived child tuple d_j is either in $\bar{D}_0$ and not deleted from $\bar{D}_0$, in $\bar{D}_0$ but deleted from $\bar{D}_0$, or not in $\bar{D}_0$. Consider case (A) for tuple t. If all derived child tuples are in $\bar{D}_0$ and not deleted from $\bar{D}_0$, then production rule r-ri generated from r is triggered by t's deletion and re-inserts t. If one or more derived child tuples d_j are in $\bar{D}_0$ but deleted from $\bar{D}_0$, or not in $\bar{D}_0$, then, by the induction hypothesis, these d_j's are inserted by rule processing. Consider the point in rule processing after the last such d_j is inserted; production rule r-ins-j generated from r is triggered by d_j's insertion and inserts t. Now consider case (B) for tuple t. Since T is not valid in $\bar{D}_0$, at least one of t's children is an inserted base tuple b_i or a derived tuple d_j whose subtree is not valid in $\bar{D}_0$. If one of t's children is an inserted base tuple b_i, then production rule r-ins-i generated from r is triggered by b_i's insertion and inserts t. If one of t's children is a derived tuple d_j whose subtree is not valid in $\bar{D}_0$, then, by the induction hypothesis, d_j is inserted by rule processing; production rule r-ins-j generated from r is triggered by d_j's insertion and inserts t. $\Box$

5 Stratified Negation

Most deductive database rule languages include *negation*, so that relations appearing on the RHS of a deductive rule can appear either *positively* (as in our initial language) or *negatively*. Using our abbreviated syntax, the general form of a deductive rule with negation is:

$$\textit{rule-name}:\ D(\bar{E}) \text{ :- } R_1(\bar{X}_1),\ \ldots,\ R_k(\bar{X}_k),\ \neg R_{k+1}(\bar{X}_{k+1}),\ \ldots,\ \neg R_n(\bar{X}_n),\ P(\bar{E}')$$

with the same requirements as given for deductive rules in Section 2, along with the following *safety* requirement [CGT90,Ull89]:

- All X's on the RHS of the rule that are variables and appear in $\bar{X}_{k+1}, \ldots, \bar{X}_n$ (i.e. in a negated relation) also appear in $\bar{X}_1, \ldots, \bar{X}_k$ (i.e. in a positive relation).

These rules have the standard deductive interpretation based on the *closed world assumption* [CGT90, Ull89]: Consider every set of tuples, one each from positive relations $R_1, \ldots, R_k$, satisfying the following conditions:

1. All RHS X's in $R_1, \ldots, R_k$ that are constants match the values in the corresponding tuple fields.

2. If all RHS X's in $R_1, \ldots, R_k$ that are variables are assigned values from the corresponding tuple fields then each variable is assigned a unique value and predicate P is satisfied.

For each set of tuples satisfying these conditions, use the variable assignments from condition 2 and the constants in $\bar{X}_{k+1}, \ldots, \bar{X}_n$ to construct one tuple for each negated relation $\neg R_{k+1}, \ldots, \neg R_n$. If each negated relation does not contain its corresponding tuple, then the tuple defined by $\bar{E}$ using the variable assignments from condition 2 is in relation D.

To guarantee unique values for derived relations, deductive database rule languages with negation often require that the set of rules is *stratified* [CGT90,Ull89]. A set of rules is stratified if the derived relations and their corresponding deductive rules can be partitioned into n *strata*, $n \geq 1$, such that:

- If a rule in stratum i has on its RHS a negated relation R, then R is either a base relation or is a derived relation belonging to a stratum j such that $j < i$.

- If a rule in stratum i has on its RHS a positive relation R, then R is either a base relation or is a derived relation belonging to a stratum j such that $j \leq i$.

The effect of stratification is that the derived relations associated with each stratum can be completely computed once the derived relations for all lower strata have been computed, i.e. the strata can be computed in ascending order.

We extend our method to handle deductive rules with stratified negation. For illustrative purposes, our running example (Section 2.1) is extended to include a third derived relation:

```
Unconnected(city1, city2)
```

Unconnected contains pairs of cities such that both cities have train stations but there is no route from the first city to the second city. *Unconnected* is defined by the following deductive rule (using **not** for $\neg$):

```
uc: Unconnected(C1,C2) :- Station(C1,S1), Station(C2,S2), not Route(C1,C2)
```

The entire set of five deductive rules is stratified, with derived relations *Route* and *Reach-Cal* (rules **rt1**, **rt2**, **rc1**, and **rc2**) in stratum 1 and derived relation *Unconnected* (rule **uc**) in stratum 2.

5.1 Extended Notation

Adding negation to deductive rules requires our notation for production rules to be extended accordingly. A rule in the extended notation is denoted by:

$$\textit{rule-name}: \ \{\textbf{ins} \mid \textbf{del}\}\ D(\bar{E}) \leftarrow$$
$$[\textbf{not}]\ [\textbf{old}]\ R_1(\bar{X}_1),\ \ldots,\ \{\textbf{ins} \mid \textbf{del}\}\ R_i(\bar{X}_i),\ \ldots,\ [\textbf{not}]\ [\textbf{old}]\ R_n(\bar{X}_n),\ P(\bar{E}')$$

i.e. each RHS relation not tagged with **ins** or **del** may be tagged with **not**. The translation from production rules in this notation to Starburst production rules proceeds as follows. A Starburst rule is generated as described in Section 3.2, ignoring all relations tagged with **not**. Then, for each relation R tagged with **not**, the following conjunct is added to the **where** clause in the *Select-Expression* of the rule action:

```
not exists (select * from [old] R where Pred)
```

where predicate *Pred* is a conjunction of equalities between attributes in R and the corresponding constants and attributes from positive relations. As an example, consider the following production rule in our extended notation:

```
uc-pi-1: ins Unconnected(C1,C2) <- ins Station(C1,S1), Station(C2,S2),
                                   not Route(C1,C2)
```

Intuitively, when a new station is inserted, this rule inserts into `unconnected` the new station's city $C1$ with all cities $C2$ such that there is no route from $C1$ to $C2$ and the pair is not already in `unconnected`. The corresponding production rule in Starburst notation is:

```
create rule uc-pi-1 on Station
when inserted
then insert into Unconnected
        select distinct inserted.city, Station.city
        from inserted, Station
        where <inserted.city,Station.city> not in (select * from Unconnected)
        and not exists
          (select * from Route
           where inserted.city = Route.city1 and Station.city = Route.city2)
```

5.2 Handling Multiple Strata

Consider a set of deductive rules for derived relations with n strata. The production rules generated from the deductive rules will be ordered so that they compute the n strata of derived relations in order. That is, for each i, $1 \leq i < n$, all production rules generated from deductive rules in stratum i are specified to precede all production rules generated from deductive rules in stratum $i+1$. Since the derived relations are materialized, the effect of this is that when production rules for a stratum i are processed, these rules can treat all derived relations from all strata $j < i$ as if they are base relations. Hence, given the requirements of stratification, we need only describe production rule generation for a single stratum, treating any RHS derived relations not in that stratum (which includes all negated derived relations) as if they are base relations.

5.3 Generating Production Rules for Each Stratum

Again, we assume initially that all derived relations and deductive rules are defined when the database is created, i.e. when the relevant base relations are empty. In Section 7.2 we explain how derived relations and deductive rules can be added when other relations are non-empty.

Consider the following generic deductive rule with negation:

$$r: \; D(\bar{E}) \;\text{:-}\; R_1(\bar{X}_1),\, \ldots,\, R_k(\bar{X}_k),\, \neg R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \neg R_n(\bar{X}_n),\, P(\bar{E}')$$

From this rule, the following $2n+1$ production rules are generated:

1. *Negated-Insertion Rules* – one rule for each i, $k+1 \le i \le n$

   $$r\text{-ni-}i: \; \textbf{del}\; D(\bar{E}) \leftarrow \textbf{old}\; R_1(\bar{X}_1),\, \ldots,\, \textbf{old}\; R_k(\bar{X}_k),\\ \textbf{not old}\; R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \textbf{ins}\; R_i(\bar{X}_i),\, \ldots,\, \textbf{not old}\; R_n(\bar{X}_n),\, P(\bar{E}')$$

2. *Positive-Deletion Rules* – one rule for each i, $1 \le i \le k$

   $$r\text{-pd-}i: \; \textbf{del}\; D(\bar{E}) \leftarrow \textbf{old}\; R_1(\bar{X}_1),\, \ldots,\, \textbf{del}\; R_i(\bar{X}_i),\, \ldots,\, \textbf{old}\; R_k(\bar{X}_k),\\ \textbf{not old}\; R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \textbf{not old}\; R_n(\bar{X}_n),\, P(\bar{E}')$$

3. *Re-insert Rule*

   $$r\text{-ri}: \; \textbf{ins}\; D(\bar{E}) \leftarrow \textbf{del}\; D(\bar{E}),\, R_1(\bar{X}_1),\, \ldots,\, R_k(\bar{X}_k),\\ \textbf{not}\; R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \textbf{not}\; R_n(\bar{X}_n),\, P(\bar{E}')$$

4. *Negated-Deletion Rules* – one rule for each i, $k+1 \le i \le n$

   $$r\text{-nd-}i: \; \textbf{ins}\; D(\bar{E}) \leftarrow R_1(\bar{X}_1),\, \ldots,\, R_k(\bar{X}_k),\\ \textbf{not}\; R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \textbf{del}\; R_i(\bar{X}_i),\, \textbf{not}\; R_i(\bar{X}_i),\, \ldots,\, \textbf{not}\; R_n(\bar{X}_n),\, P(\bar{E}')$$

5. *Positive-Insertion Rules* – one rule for each i, $1 \le i \le n$

   $$r\text{-pi-}i: \; \textbf{ins}\; D(\bar{E}) \leftarrow R_1(\bar{X}_1),\, \ldots,\, \textbf{ins}\; R_i(\bar{X}_i),\, \ldots,\, R_k(\bar{X}_k),\\ \textbf{not}\; R_{k+1}(\bar{X}_{k+1}),\, \ldots,\, \textbf{not}\; R_n(\bar{X}_n),\, P(\bar{E}')$$

The rules are partially ordered so that rules in each group have priority over rules in the next group.

To understand the behavior of these rules, first observe what triggers rules in each group and what operations the rule actions perform:

1. triggered by insertions into negated R, deletes from D
2. triggered by deletions from positive R, deletes from D
3. triggered by deletions from D, inserts into D
4. triggered by deletions from negated R, inserts into D
5. triggered by insertions into positive R, inserts into D

Since there can be no recursion involving negated relations (due to stratification), we see that execution of a rule in a group i cannot trigger a rule in a group $j < i$. Hence, rule processing consists of one phase per group of rules: In phase 1, tuples are deleted from the derived relation based on insertions into negated relations. In phase 2, tuples are deleted from the derived relation based on deletions from positive relations. In phase 3, tuples with multiple derivations are re-inserted into the derived relation (as explained in Section 4.1). In phase 4, tuples are inserted into the derived relation based on deletions from negated relations. Finally, in phase 5, tuples are inserted into the derived relation based on insertions into positive relations. Notice that in the negated-deletion rules (group 4), there is a **not** $R_i(\bar{X}_i)$ clause in addition to **del** $R_i(\bar{X}_i)$. This is present to eliminate cases in which a tuple is deleted from and then re-inserted into R_i, either by the user or by rules computing derived relations in a lower stratum.[9]

Multiple deductive rules are handled as in Section 4.1: Production rules are generated as described above for each deductive rule. The entire set of rules is partially ordered so that all rules in group i for any deductive rule precede all rules in group $i+1$ for any deductive rule. Again, there is no need to order rules within each group, but recall that rules across strata are ordered so that all rules for a given stratum have priority over all rules for the next stratum.

5.4 Example

Consider the deductive rule with negation introduced above for derived relation *Unconnected*:

```
uc: Unconnected(C1,C2) :- Station(C1,S1), Station(C2,S2), not Route(C1,C2)
```

The following production rules are generated:

```
uc-ni-1: del Unconnected(C1,C2) <- old Station(C1,S1), old Station(C2,S2),
                                   ins Route(C1,C2)

uc-pd-1: del Unconnected(C1,C2) <- del Station(C1,S1), old Station(C2,S2),
                                   not old Route(C1,C2)
uc-pd-2: del Unconnected(C1,C2) <- old Station(C1,S1), del Station(C2,S2),
                                   not old Route(C1,C2)

uc-ri:   ins Unconnected(C1,C2) <- del Unconnected(C1,C2),
                                   Station(C1,S1), Station(C2,S2),
                                   not Route(C1,C2)

uc-nd-1: ins Unconnected(C1,C2) <- Station(C1,S1), Station(C2,S2),
                                   del Route(C1,C2), not Route(C1,C2)

uc-pi-1: ins Unconnected(C1,C2) <- ins Station(C1,S1), Station(C2,S2),
                                   not Route(C1,C2)
uc-pi-2: ins Unconnected(C1,C2) <- Station(C1,S1), ins Station(C2,S2),
                                   not Route(C1,C2)
```

These production rules are ordered so that all rules in each group have priority over all rules in the next group. The corresponding rules in Starburst notation are given in Appendix A.2. These production rules are used together with the production rules for derived relations *Route* and *Reach-Cal* in Section 4.2, with these rules corresponding to stratum 2 and the rules for *Route* and *Reach-Cal* corresponding to stratum 1. Hence, these rules are specified to have lower priority than the rules for *Route* and *Reach-Cal*.

[9]In Starburst, although the net effect of an insertion followed by a deletion is nothing, the net effect of a deletion followed by an insertion is a deletion and an insertion; see [WF90].

5.5 Correctness

The correctness proof is similar to Theorem 4.1. As explained in Section 5.2, we consider each stratum separately (i.e. the proof itself considers only one stratum), with derived relations from lower strata treated as base relations. Hence, we assume that negation is over base relations only. In the proof, the notion of a derivation tree is extended so that each leaf represents either the presence of a base relation tuple, denoted b, or the absence of a base relation tuple, denoted $\neg b$.

Theorem 5.1 Consider arbitrary derived relations $D_1, \ldots, D_m$ defined over arbitrary base relations $B_1, \ldots, B_n$ by arbitrary deductive rules $r_1, \ldots, r_x$ that may include negation over base relations. Let $\rho_1, \ldots, \rho_y$ be the production rules generated from $r_1, \ldots, r_x$ as described in Section 5.3. Suppose $D_1, \ldots, D_m$ are consistent initially with $B_1, \ldots, B_n$, then arbitrary tuples are deleted from and/or inserted into $B_1, \ldots, B_n$, then rule processing is invoked with $\rho_1, \ldots, \rho_y$. When rule processing terminates, $D_1, \ldots, D_m$ are consistent with the new values of $B_1, \ldots, B_n$.

Proof: As in Theorem 4.1, let $\bar{B}_0$ denote $B_1, \ldots, B_n$ before the modifications, let $\bar{B}$ denote $B_1, \ldots, B_n$ after the modifications, let $\bar{D}_0$ denote $D_1, \ldots, D_m$ before rule processing, and let $\bar{D}$ denote $D_1, \ldots, D_m$ consistent with $\bar{B}$. By assumption, $\bar{D}_0$ is consistent with $\bar{B}_0$ and $\bar{D}$ is consistent with $\bar{B}$; we must show that rule processing modifies $D_1, \ldots, D_m$ from $\bar{D}_0$ to $\bar{D}$. In the first step we show that any tuple that is in $\bar{D}_0$ but not in $\bar{D}$ is deleted by rule processing; in the second step we show that any tuple that is in $\bar{D}$ but is not in $\bar{D}_0$, or that is in $\bar{D}$ but is deleted from $\bar{D}_0$ by rule processing, is inserted by rule processing. It is straightforward to see that rule processing cannot insert tuples that are not in $\bar{D}$, so these two steps complete the proof.

[1] *Deletions*: Let t be a tuple in $\bar{D}_0$ but not in $\bar{D}$. Then any derivation tree for t in $\bar{D}_0$ is not valid in $\bar{D}$; consider one such tree T. We prove by induction on the depth of T that t is deleted by rule processing.

Base case – T has depth 1. Let t's children in T be base tuples $b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_l$, with t derived from its children by deductive rule r. Since T is not valid in $\bar{D}$, at least either one b_i has been deleted or one $\neg b_j$ has been inserted. Hence, either production rule r-pd-i generated from r is triggered by b_i's deletion and deletes t, or production rule r-ni-j generated from r is triggered by b_j's insertion and deletes t.

Induction – T has depth $n > 1$. Let t's children in T be base and derived tuples $t_1, \ldots, t_k$, $\neg b_{k+1}, \ldots, \neg b_l$, with t derived from its children by deductive rule r. If T is not valid in $\bar{D}$ because one of $\neg b_{k+1}, \ldots, \neg b_l$ has been inserted (call it b_i), then production rule r-ni-i generated from r is triggered by b_i's insertion and deletes t. Otherwise, T is not valid in $\bar{D}$ because one of $t_1, \ldots, t_k$ is a deleted base tuple b_i or a derived tuple d_j whose subtree is not valid in $\bar{D}$, so the proof proceeds exactly as in the induction step for [1] in Theorem 4.1.

[2] *Insertions*: Let t be a tuple in $\bar{D}$ that is either (A) deleted from $\bar{D}_0$ by rule processing or (B) not in $\bar{D}_0$. Consider any derivation tree T for t in $\bar{D}$. We prove by induction on the depth of T that t is inserted by rule processing.

Base case – T has depth 1. Let t's children in T be base tuples $b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_l$, with t derived from its children by deductive rule r. In case (A), production rule r-ri generated from r is triggered by t's deletion and re-inserts t. In case (B), since T is not valid in $\bar{D}_0$, at least either one b_i has been inserted or one $\neg b_j$ has been deleted (and not re-inserted). Hence, either production rule r-pi-i generated from r is triggered by b_i's insertion and inserts t or production rule r-nd-i generated from r is triggered by b_j's deletion and inserts t.

Induction – T has depth $n > 1$. Let t's children in T be base and derived tuples $t_1, \ldots, t_k$, $\neg b_{k+1}, \ldots, \neg b_l$, with t derived from its children by deductive rule r. If one of $\neg b_{k+1}, \ldots, \neg b_l$ has been deleted (call it b_i), then production rule r-nd-i generated from r is triggered by b_i's deletion and inserts t. Otherwise, the proof proceeds exactly as in the induction step for [2] in Theorem 4.1. $\square$

6 Simplifications for Unique Derivations

The "re-insert" rules generated in our approach are due to the fact that tuples in derived relations may have multiple derivations. Suppose that for a derived relation D we know that each tuple in D is guaranteed to have only one derivation. Then re-insert rules need not be generated from the deductive rules for D since the re-insert rules will never insert any tuples. As a second simplification for unique derivations, **distinct** can be eliminated from the **insert** *Select-Expressions* generated in rule actions since duplicates will never be produced.

The notion of a derived relation with unique derivations corresponds directly to the notion of *safe* materialized views in [CW91].[10] In [CW91], static criteria are given for determining whether an SQL view is safe based on the view definition and key information for base relations. In the following, we show how these criteria can be adapted to determine whether a derived relation has unique derivations. Our direct adaptation applies only to derived relations defined by one non-recursive deductive rule; extending the criteria for multiple and recursive rules is left as future work. In addition, for brevity we do not consider negation here, although criteria in [CW91] for negatively nested subqueries can be adapted for negation.

Let D be a derived relation defined by deductive rule r without recursion or negation. Let the *bound attributes* (call them B) of r be computed as follows:

1. Initialize B to contain all attributes from RHS relations whose place corresponds to a variable also appearing on the LHS of r (or that is joined to a LHS variable by equality in the predicate).

2. Add to B all attributes from RHS relations whose place corresponds to a constant (or to a variable equated to a constant in the predicate).

[10] In general, when views are safe (or, equivalently, when derived relations have unique derivations) this is an indication that a materialized approach can be realized efficiently.

3. Repeat until B is unchanged: (a) Add to B all attributes from RHS relations whose place corresponds to a variable that also corresponds to an attribute already in B (or that is joined to some attribute already in B by equality in the predicate). (b) Add to B every attribute of any RHS relation such that B includes a key for that relation.

Based on results in [CW91], derived relation D is guaranteed to have unique derivations if the bound attributes of r include a key for every relation on the RHS of r.

7 Derived Relations on Demand

We now consider the non-materialized case, where derived relations are computed only as needed in order to execute given queries. In the simplest approach, when a query is submitted that references derived relations, the derived relations are materialized in their entirety from the base relations, the query is executed, then the derived relations are deleted. In more sophisticated approaches, only those portions of the derived relations needed to execute the query are computed. We first consider the simple approach; more sophisticated approaches are discussed in Section 7.1.

Suppose a query Q is submitted that references non-materialized derived relations $D_1, \ldots, D_k$, where $D_1, \ldots, D_k$ are defined by a set of (stratified) deductive rules over base relations $B_1, \ldots, B_n$ and additional derived relations $D_{k+1}, \ldots, D_m$. To execute the query, we must materialize the derived relations $D_1, \ldots, D_k$, which also requires materializing $D_{k+1}, \ldots, D_m$. In order for the production rules to know that $D_1, \ldots, D_m$ are the derived relations to be materialized, we maintain a special relation called **Materialize** (say) in which $D_1, \ldots, D_m$ are explicitly named. (Other mechanisms for knowing which relations must be materialized are possible; see below.) When the query processor receives query Q, it inserts into **Materialize** tuples that name relations $D_1, \ldots, D_m$, then it invokes rule processing.[11] When rule processing completes, query Q is executed, then the contents of relation **Materialize** and of derived relations $D_1, \ldots, D_m$ are deleted. Note that this approach can be very inefficient, since the entire contents of derived relations $D_1, \ldots, D_m$ are computed before processing query Q. However, if Q contains selection conditions then this approach can be made more efficient using various known optimizations; see Section 7.1.

Consider our standard generic deductive rule, including negation:

$$r: \ D(\bar{E}) \text{ :- } R_1(\bar{X}_1), \ \ldots, \ R_k(\bar{X}_k), \neg R_{k+1}(\bar{X}_{k+1}), \ \ldots, \ \neg R_n(\bar{X}_n), \ P(\bar{E}')$$

The identical production rules are generated from r as for the materialized approach (Section 5.3), and one additional rule "*r-start*" is generated. The role of *r-start* is to perform one application of deductive rule r, inserting initial tuples into D based on existing relations. These insertions will trigger the other production rules as necessary to complete the computation of D. Hence, *r-start* is triggered by insertions into relation **Materialize**, and it has a condition that checks whether a tuple naming D has been inserted. The action of *r-start* is an **insert** operation that intuitively corresponds to:

[11] For efficiency, rule processing should be invoked for only those rules that compute $D_1, \ldots, D_m$.

$$r\text{-start}: \ \mathbf{ins}\ D(\bar{E}) \leftarrow R_1(\bar{X}_1),\ \ldots,\ R_k(\bar{X}_k),\ \neg R_{k+1}(\bar{X}_{k+1}),\ \ldots,\ \neg R_n(\bar{X}_n),\ P(\bar{E}')$$

That is, r-*start*'s action is the same as the **insert** operation generated in insertion production rules for r (Sections 3.2 and 5.1), except no transition tables are used and the **not in** clause is omitted.

For multiple deductive rules, production rules as described in Section 5.3 along with an additional *start* rule are generated for each one. The production rules are ordered as described in Section 5.3, and all *start* rules are given highest priority.

We outline the correctness argument for this approach; full details are omitted. Let Q be a query that references derived relations $\bar{D}$, let $\bar{r}$ be the deductive rules for $\bar{D}$, and let $\bar{B}$ be the relevant base relations at the time Q is submitted. We must show that execution of the *start* rules for $\bar{r}$ followed by execution of the other production rules generated from $\bar{r}$ materialize $\bar{D}$ consistent with $\bar{B}$. To see this, consider the following hypothetical transaction τ and the materialized approach: At the start of τ, all base and derived relations are empty. In the "user" portion of τ, tuples are inserted to bring the relevant base relations to their state in $\bar{B}$. Production rules generated from $\bar{r}$ are then processed to modify derived relations $\bar{D}$ according to the insertions into $\bar{B}$. By our results in Section 5.5, at the end of τ, derived relations $\bar{D}$ are consistent with $\bar{B}$. Now consider query Q. When Q is submitted, the derived relations are empty. The key observation is that the *start* rules insert into the derived relations exactly the same data that is inserted by the production rules in τ responding to τ's base relation insertions. Following these initial insertions, rule processing in τ and for query Q are identical. Hence, when rule processing for query Q terminates, derived relations $\bar{D}$ are consistent with $\bar{B}$.

While we have suggested the use of relation **Materialize** and *start* rules to initiate computation of the derived relations, other approaches are possible. For example, the query processor itself could execute the **insert** operations corresponding to the actions of our *start* rules, then invoke rule processing. If the database production rule language includes rules triggered by **select** operations (as in, e.g., POSTGRES [SJGP90]) and has sufficient flexibility, then rule processing to compute derived relations could be invoked directly in response to a query, without the intervention of the query processor.

It is interesting to note that the behavior of production rule execution in our non-materialized approach computes derived relations using a strategy very similar to semi-naive evaluation. The computation is not identical, however, because the production rules are not totally ordered and only one rule is applied in each "iteration" rather than all rules.

7.1 Derived Relations with Selection Conditions

One important advantage of a non-materialized approach to deductive databases is that when a query references derived relations, and when the query includes a selection condition, then based on the selection condition it may be sufficient to compute only a portion of the derived data. For example, if a query requests all tuples of a derived relation D such that $D.attr = 5$, it only is necessary to compute enough of D to guarantee that all tuples with $D.attr = 5$ are produced. A number of methods have been developed for computing partial derived data based on selection conditions, most notably the *magic sets* technique [BR86]. Most of these methods rewrite the deductive rules based on the selection condition, then evaluate the new set of deductive rules using a standard mechanism such as semi-naive evaluation. Such rewriting approaches can be used to optimize our method for non-materialized derived relations: When a query with a selection condition is submitted, the deductive rules are rewritten using, e.g., magic sets, then production rules are generated from the new deductive rules. This requires generating production rules for each query pattern (or *adornment*, see [Ull89]). The production rules are parameterized, then instantiated for a given query's selection condition by replacing the parameters with appropriate constant values.

7.2 Non-Empty Relations in the Materialized Approach

In our materialized approach, we assumed that production rules generated from deductive rules would be installed when the database is created, i.e. when all base and derived relations are empty. However, after the database has been created, a user may want to define new derived relations and deductive rules based on existing base and/or derived relations. To do this, a mechanism very similar to our approach for non-materialized derived relations can be used to bring the new materialized derived relations "up to date" with existing relations.

Suppose a new deductive rule r is defined for derived relation D. Production rules are generated from r as described in Section 5.3. After the rules are installed, one **insert** operation is executed—the same operation that is generated in the action of rule *r-start* in the non-materialized approach. If the user adds multiple new deductive rules (for one or more new derived relations), then one **insert** operation is executed for each new deductive rule. These insertions trigger the new production rules as necessary to ensure that, at the end of rule processing, the new derived relations are consistent with existing relations.[12] The correctness argument for this approach directly parallels the correctness argument given above for non-materialized derived relations. Note that this mechanism also can be used to add new deductive rules for existing derived relations.

8 Conclusions and Future Work

We have described a complete framework whereby the production rule mechanism provided by an active database system can be used to support a deductive database system. Derived relations are specified using deductive rules with built-in predicates and stratified negation; the deductive rules are compiled automatically into production rules. In our materialized approach, derived relations are stored in the database, and production rules maintain the consistency of derived relations with base relations. In our non-materialized approach, production rules compute derived relations as needed by queries. In both approaches, derived relations are computed incrementally.

[12] For concurrency control purposes, one transaction should be executed that first creates the new derived relations and production rules, then performs the **insert** operations; this ensures consistency even in the presence of other transactions [WCL91].

Both approaches provide a mechanism for quickly and effectively supporting deductive rules in any environment that provides production rules. We are currently realizing this work in the context of Esprit Project *Idea*. The project includes a prototype whose core database engine supports only production rules; derived data is specified using a Datalog-like language that is translated automatically into production rules [CM93].

Our framework generates correct production rules for any set of deductive rules. However, for certain classes of deductive rules it is possible to generate more efficient production rules. An example of this is described in Section 6, where key information is used to determine when derived relations have unique derivations. Future work includes exploring other optimizations based on key information, such as combining or eliminating production rules, and removing extraneous **where** clauses from rule actions.

Our work has concentrated on a specific method for computing derived data. As discussed in Section 1.1, there are a number of alternative methods: derivations may be counted (such as in [NY83, GMS93]), other auxiliary information may be maintained (such as in [DT92, Kuc91, WDSY91]), or alternative derivations may be checked before derived tuples are deleted (such as in [UO92]). All of these methods are aimed at avoiding the "re-insert" approach we have used (recall Sections 4.1 and 5.3). A useful future contribution would be to encode several different methods using production rules and study their relative performance.

Throughout the paper we have treated updates as deletes followed by inserts. We plan to modify our approach to treat updates directly. In general, updates to base relations still will result in deletes followed by inserts to derived relations, as in [CW91]. Modifying our framework for this is straightforward—it consists of adding productions rules triggered by updates, and using transition tables **old-updated** and **new-updated** along with **deleted** and **inserted**. However, by considering key information, in some cases we may be able to generate production rules that propagate updates on base relations directly as updates on derived relations.

In this paper we have considered two distinct approaches to derived data: materialized and non-materialized. In the materialized approach all derived data is maintained at all times, while in the non-materialized approach no derived data is maintained. We are interested in exploring an intermediate approach: Initially, derived data is not materialized, but when derived data is materialized in response to queries it is not deleted. Existing derived data may or may not be kept consistent with base data, and new derived data is not computed from new base data until it is needed. This represents a "lazy" approach to materialization, and could be used for deductive data as well as for other derived data such as conventional views.

A Appendix

A.1 Starburst Rules for Section 4.2

```
create rule rt1-del-1 on Train
when deleted
then delete from Route
     where <city1,city2> in (select city1, city2 from deleted)

create rule rt2-del-1 on Route
when deleted
then delete from Route where <city1,city2> in
       (select deleted.city1, Route.city2 from deleted, old Route
        where deleted.city2 = Route.city1)

create rule rt2-del-2 on Route
when deleted
then delete from Route where <city1,city2> in
       (select Route.city1, deleted.city2 from old Route, deleted
        where Route.city2 = deleted.city1)

create rule rc1-del-1 on Station
when deleted
then delete from Reach-Cal where city in
       (select city from deleted where state = "California")

create rule rc2-del-1 on Route
when deleted
then delete from Reach-Cal where city in
       (select deleted.city1 from deleted, old Reach-Cal
        where deleted.city2 = Reach-Cal.city

create rule rc2-del-2 on Reach-Cal
when deleted
then delete from Reach-Cal where city in
       (select Route.city1 from Route, deleted
        where Route.city2 = deleted.city)

create rule rt1-ri on Route
when deleted
then insert into Route
        select distinct deleted.city1, deleted.city2 from deleted, Train
        where deleted.city1 = Train.city1
          and deleted.city2 = Train.city2
          and <deleted.city1,deleted.city2> not in (select * from Route)
follows rt1-del-1, rt2-del-1, rt2-del-2, rc1-del-1, rc2-del-1, rc2-del-2

create rule rt2-ri on Route
when deleted
then insert into Route
        select distinct deleted.city1, deleted.city2
        from deleted, Route R1, Route R2
        where deleted.city1 = R1.city1
```

```
          and deleted.city2 = R2.city2
          and R1.city2 = R2.city1
          and <deleted.city1,deleted.city2> not in (select * from Route)
follows rt1-del-1, rt2-del-1, rt2-del-2, rc1-del-1, rc2-del-1, rc2-del-2

create rule rc1-ri on Reach-Cal
when deleted
then insert into Reach-Cal
        select distinct deleted.city from deleted, Station
        where deleted.city = Station.city
          and Station.state = "California"
          and deleted.city not in (select * from Reach-Cal)
follows rt1-del-1, rt2-del-1, rt2-del-2, rc1-del-1, rc2-del-1, rc2-del-2

create rule rc2-ri on Reach-Cal
when deleted
then insert into Reach-Cal
        select distinct deleted.city from deleted, Route, Reach-Cal
        where deleted.city = Route.city1
          and Route.city2 = Reach-Cal.city
          and deleted.city not in (select * from Reach-Cal)
follows rt1-del-1, rt2-del-1, rt2-del-2, rc1-del-1, rc2-del-1, rc2-del-2

create rule rt1-ins-1 on Train
when inserted
then insert into Route
        select distinct city1, city2 from inserted
        where <city1,city2> not in (select * from Route)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri

create rule rt2-ins-1 on Route
when inserted
then insert into Route
        select distinct inserted.city1, Route.city2 from inserted, Route
        where inserted.city2 = Route.city1
          and <inserted.city1,Route.city2> not in (select * from Route)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri

create rule rt2-ins-2 on Route
when inserted
then insert into Route
        select distinct Route.city1, inserted.city2 from Route, inserted
        where Route.city2 = inserted.city1
          and <Route.city1,inserted.city2> not in (select * from Route)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri

create rule rc1-ins-1 on Station
when inserted
then insert into Reach-Cal
        select distinct city from inserted
        where state = "California"
          and city not in (select * from Reach-Cal)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri
```

```
create rule rc2-ins-1 on Route
when inserted
then insert into Reach-Cal
        select distinct inserted.city1 from inserted, Reach-Cal
        where inserted.city2 = Reach-Cal.city
          and inserted.city1 not in (select * from Reach-Cal)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri

create rule rc2-ins-2 on Reach-Cal
when inserted
then insert into Reach-Cal
        select distinct Route.city1 from Route, inserted
        where Route.city2 = inserted.city
          and Route.city1 not in (select * from Reach-Cal)
follows rt1-ri, rt2-ri, rc1-ri, rc2-ri
```

A.2 Starburst Rules for Section 5.4

```
create rule uc-ni-1 on Route
when inserted
then delete from Unconnected where <city1,city2> in
       (select S1.city, S2.city from old Station S1, old Station S2, inserted
        where S1.city = inserted.city1 and S2.city = inserted.city2)
follows rt1-ins-1, rt2-ins-1, rt2-ins-2, rc1-ins-1, rc2-ins-1, rc2-ins-2

create rule uc-pd-1 on Station
when deleted
them delete from Unconnected where <city1,city2> in
       (select deleted.city, Station.city from deleted, old Station
        where not exists
          (select * from old Route
           where deleted.city = Route.city1 and Station.city = Route.city2))
follows uc-ni-1

create rule uc-pd-2 on Station
when deleted
them delete from Unconnected where <city1,city2> in
       (select Station.city, deleted.city from old Station, deleted
        where not exists
          (select * from old Route
           where Station.city = Route.city1 and deleted.city = Route.city2))
follows uc-ni-1


create rule uc-ri on Unconnected
when deleted
then insert into Unconnected
        select distinct deleted.city1, deleted.city2
        from deleted, Station S1, Station S2
        where deleted.city1 = S1.city and deleted.city2 = S2.city
        and <deleted.city1,deleted.city2> not in (select * from Unconnected)
        and not exists
          (select * from Route
```

```
           where deleted.city1 = Route.city1 and deleted.city2 = Route.city2)
follows uc-pd-1, uc-pd-2

create rule uc-nd-1 on Route
when deleted
then insert into Unconnected
        select distinct S1.city, S2.city from Station S1, Station S2, deleted
        where S1.city = deleted.city1 and S2.city = deleted.city2
        and <S1.city,S2.city> not in (select * from Unconnected)
        and not exists
          (select * from Route
           where S1.city = Route.city1 and S2.city = Route.city2)
follows uc-ri

create rule uc-pi-1 on Station
when inserted
then insert into Unconnected
        select distinct inserted.city, Station.city from inserted, Station
        where <inserted.city,Station.city> not in (select * from Unconnected)
        and not exists
          (select * from Route
           where inserted.city = Route.city1 and Station.city = Route.city2)
follows uc-nd-1

create rule uc-pi-2 on Station
when inserted
then insert into Unconnected
        select distinct Station.city, inserted.city from Station, inserted
        where <Station.city,inserted.city> not in (select * from Unconnected)
        and not exists
          (select * from Route
           where Station.city = Route.city1 and inserted.city = Route.city2)
follows uc-nd-1
```

Acknowledgements

Thanks to Bobbie Cochrane and Elena Baralis for helpful comments on an initial draft, and to Stefano Paraboschi for useful discussions.

References

[AP87] K.R. Apt and J.-M. Pugin. Maintenance of stratified databases viewed as a belief revision system. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 136–145, San Diego, California, March 1987.

[BJ93] P. Bayer and W. Jonker. A framework for supporting triggers in deductive databases. In *Proceedings of the First International Workshop on Rules in Database Systems*, Edinburgh, Scotland, August 1993.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 1986.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 225–236, Atlantic City, New Jersey, May 1990.

[Cer92] S. Ceri. A declarative approach to active databases. In *Proceedings of the Eighth International Conference on Data Engineering* (invited paper), pages 452–456, Phoenix, Arizona, February 1992.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. To appear in *ACM Transactions on Database Systems*, 1994.

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

[CM93] S. Ceri and R. Manthey. First specification of Chimera. Technical Report IDEA.DD.2P.004.02, Politecnico di Milano, Milan, Italy, May 1993.

[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.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[DHW94] U. Dayal, E.N. Hanson, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press, New York, 1994.

[DT92] G. Dong and R. Topor. Incremental evaluation of Datalog queries. In *Proceedings of the 1992 International Conference on Database Theory*, Berlin, Germany, October 1992.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[H+90] L.M. Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HD93a] J.V. Harrison and S.W. Dietrich. Condition monitoring in an active deductive database. *Journal of Information Systems*, 1993.

[HD93b] J.V. Harrison and S.W. Dietrich. Integrating active and deductive rules. In *Proceedings of the First International Workshop on Rules in Database Systems*, pages 288–305, Edinburgh, Scotland, August 1993.

[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.

[KdMS90] J. Kiernan, C. de Maindreville, and E. Simon. Making deductive databases a practical technology: A step forward. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 237–246, Atlantic City, New Jersey, May 1990.

[Kuc91] V. Kuchenhoff. On the efficient computation of the difference between consecutive database states. In *Proceedings of the 1991 International Conference on Deductive and Object-Oriented Databases*, pages 478–502, Munich, Germany, December 1991.

[Min88] J. Minker. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, San Mateo, California, 1988.

[Nic82] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[NY83] J.-M. Nicolas and K. Yazdanian. An outline of DBGEN: A deductive DBMS. In *Proceedings of the IFIP World Conference*. North Holland, 1983.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, Atlantic City, New Jersey, May 1990.

[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.

[Tan91] L. Tanca. (Re-)Action in deductive databases. In *Proceedings of the Second International Workshop on Intelligent and Cooperative Information Systems*, pages 55–61, Como, Italy, October 1991.

[UKN92] S. Urban, A.P. Karadimce, and R.B. Nannapaneni. The implementation and evaluation of integrity maintenance rules in object-oriented databases. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 564–572, Phoenix, Arizona, February 1992.

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

[UO92] T. Urpi and A. Olive. A method for change computation in deductive databases. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 225–237, Vancouver, British Columbia, August 1992.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.

[WDSY91] O. Wolfson, H.M. Dewan, S.J. Stolfo, and Y. Yemini. Incremental evaluation of rules and its relationship to parallelism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 78–87, Denver, Colorado, May 1991.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

[Wid91] J. Widom. Deduction in the Starburst production rule system. IBM Research Report RJ 8135, IBM Almaden Research Center, San Jose, California, May 1991.

[Wid92] J. Widom. The Starburst Rule System: Language design, implementation, and applications. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):15–18, December 1992.