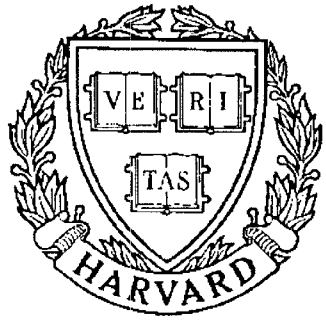
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Update Dependencies in the Relational Model

by L. Mark, N. Roussopoulos and R. Cochrane

TR 91-94

Update Dependencies in the Relational Model *

Leo Mark, Nick Roussopoulos and Roberta Cochrane

Department of Computer Science
University of Maryland
College Park, Maryland 20742
leo@cs.umd.edu, nick@cs.umd.edu, bobbie@cs.umd.edu

Abstract

Relational database systems suffer from the lack of a rich update language. In this paper we present the Update Dependency Language which allows the database designer to specify a procedure for each update that is activated when attempts are made to perform the update. Each procedure integrates the update dependencies for an update and provides an operational semantics for the update which is maintained by the system. We provide a formal definition for this language, illustrate its use, and discuss concurrency control issues related to integrating such a language into a database system.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages - *data manipulation languages (DML)*; H.2.4 [**Systems**]: Concurrency - *transaction processing*
General Terms: Design, Languages
Additional Keywords and Phrases: Database Update Specifications, Integrity, Consistency, View Update, Active Databases

1 Introduction

It is a well-recognized fact that the semantics for retrieval in the relational model is well-defined and polished, but it is missing a rich update language. Such an update language must provide mechanisms for maintaining integrity constraints, functional dependencies, and inter-relational dependencies during database update. Additionally, it should provide the ability to specify view update policies.

In most commercial systems, applications compensate for the lack of a rich update language by chaining updates together in transactions [Gra78] which are embedded in an application program. Every application programmer is required to understand and specify the logic for enforcing the dependencies between updates. This logic will be replicated in each application program that updates the database, and all of these application programs

*This research was sponsored partially by the National Science Foundation under Grant IRI-8719458, by the Air Force Office for Scientific Research under Grant AFOSR-89-0303, by Systems Research Center and by the University of Maryland Institute of Advanced Computer Studies.

must be updated to reflect all changes in the logic. This duplication and maintenance of code is costly and prone to error.

We propose an update language that is based on the concept of update dependencies. An *update dependency* is a requirement that an update cannot be performed on a database state that satisfies a given condition unless another update is also performed. For example, the requirement that an update U cannot be performed on a database state that satisfies condition C unless update T is also performed is an update dependency. We refer to U as the owner of the update dependency since it is the dependent update, and we say the pair (C, T) is an update dependency of U.

A mechanism that supports the specification and enforcement of update dependencies provides a general framework for enforcing several well-known categories of dependencies between data in the database, including integrity constraints and functional dependencies. It can also be used to record application-specific knowledge. This type of knowledge may provide information about how a violation of a dependency should be resolved. It may also be used to define and maintain policies for view update, view materialization and data evolution.

Integrity constraints support database consistency through a specification of the correct states of the database. However, this technique fails to capture the action that must be performed when an integrity constraint is violated. For example, a referential integrity constraint between suppliers and shipments in the suppliers-parts-shipments database states that `suppliers` for all `shipments` must exist in the `suppliers` relation. It does not specify what action should be performed if an attempt is made to delete a `supplier` that is referenced by at least one `shipment`. [1] With update dependencies, one can specify the consistent evolution of a database. The above referential integrity constraint can be expressed by two update dependencies:

1. If a `supplier` is deleted from the `suppliers` relation, then all `shipments` that depend on the `supplier` must also be deleted. Alternatively, we could specify that all `shipments` that depend on the deleted supplier be moved to a temporary relation for later examination by the database administrator (DBA). A third alternative might prevent deletions of a `supplier` if there exists any `shipment` that depends on it.

2. If a `shipment` is inserted with a `supplier` that is not in the `suppliers` relation, then

[1] We consider here the pure definition of referential integrity. Several systems actually allow the user to choose one of three actions to be performed when referential integrity is violated by deletion of a tuple in the parent relation. Our language allows the user to specify an arbitray action for referential integrity violations caused by insertions and updates as well as insertions.

the `supplier` is inserted, or the `shipment` is placed in a temporary relation for later examination. Alternatively, such updates may not be allowed.

This style of formulating referential integrity specifies how the database can avoid consistency violations by taking actions that are specific to an application as opposed to an all out rejection of updates if an integrity violation occurs.

Normalization [Cod72,Cod74,Arm74,Ber76,Fag77,Fag79] is a design technique that was introduced to eliminate inherent functional dependencies that result from the existence of redundancy. Although we do not advocate the use of our language as a replacement for normalization, it provides an alternative to enforcing functional dependencies when it is undesireable to normalize. Additionally, it can be used to enforce inter-relational dependencies that cannot be modeled in normal form representations.

Our language can also be used for specifying policies for updating views. The database designer often knows how a view should be updated, even when the view is not theoretically updatable [FC85,BS81]. This application-specific knowledge cannot currently be captured in the view definition. It can, however, be stated as update dependencies between updates on the view and updates on the base relations that are used to derive the view. For example, suppose an insurance database consists of the base relations `accounts(ins-no, payer)` and `insured(ins-no, patient)`, and the view `dep(payer, patient)` in which `dep.patient` is covered by an insurance policy paid by `dep.payer`. From the perspective of the DBMS it is unclear how to delete a tuple from the view `dep`; this tuple can be deleted either by deleting the `payer` from the `accounts` relation or by deleting the `patient` from the `insured` relation. However, there could be an application specific policy that says that deleting a tuple from the `dependents` relation can be performed by deleting the `patient` from the `insured` relation (see example 3). Such a policy is expressible with update dependencies.

More recent research efforts have proposed using such concepts as triggers [Esw76], alerters [BC76], condition-actions [HM76], event procedures [BFM79], and database production rule systems [Coh89, DE89, Han89, MD89, SLR88, SJGP90, WF90] to specify dependencies between data. These systems provide a facility for specifying and enforcing update dependencies. Even though several of these systems have a well-defined semantics for their rules, they do not provide a deterministic behavior when several rules trigger simultaneously. The conflict resolution mechanism, which sometimes allows user-defined priorities, determines which rule is considered first for execution. It is a current research issue how to provide conflict resolution strategies that have a deterministic semantics and encorporates user-defined priorities [ACL91,SHP88,YI89].

In the above systems, the update dependencies that must be enforced are specified by a collection of triggers or production rules. The designer specifies one rule for each update

dependency of an update. When the update occurs, all these rules are triggered. The system attempts to run one rule at a time until all of the rules are run or until one of the rules performs an action that nullifies the update.

This paper describes the *Update Dependency Language*, which also supports the specification of update dependencies. The first version of this language was introduced in [Mar85]. In contrast to production systems, this language encourages the designer to integrate the update dependencies for a given update into a procedure which is activated when attempts are made to perform the update. Although such a procedure is composed of several rules, the semantics of the language specifies that exactly one of these rules will be applied to any given update. Hence, the DBA must integrate all the updates that apply to a given state of the database into one rule, explicitly stating the order in which update dependencies are applied. Each rule in a procedure defines a transition which will transform a valid database state satisfying the condition of the rule into another valid database state when the update is requested. Each database update is transformed into an update request which is granted only if a correct transition can be found.

This paper proceeds as follows. Section 2 formally presents the syntax and semantics of the Update Dependency Language. We describe the general format of the procedures of this language and define a safeness criterion for them. Section 3 demonstrates the use of the Update Dependency Language to encode several application specific solutions to well-known problems. In section 4 we present two different strategies for executing the procedures of this language, and in section 5 we discuss issues related to integrating these executions into a traditional database environment. Section 6 discusses work that is related to or has inspired our work, and finally, section 7 contains conclusions and directions for future research.

2 The Update Dependency Language

The Update Dependency Language provides support for the operational specification of updates in relational database systems (RDBMSs). The data model in these systems is the relational model [Cod70,Cod79], which supports the definition of named relations and views. Each relation has a schema which defines its name and the name and domain of each of its attributes. Each element (or row) of a relation is a tuple. Several database systems allow duplication and hence support tables rather than relations. We make no assumptions with respect to this matter.

Users query and modify relational databases through data manipulation languages. The standard database languages, such as SQL and QUEL, perform set-oriented operations,

while other paradigms, such as Prolog, manipulate the database with a tuple-at-a-time semantics. Queries and operations are typically grouped into transactions which have the notion of atomicity. If the transaction commits then all the operations are performed; if the transaction fails then all the operations are rolled back.

With our language, the database designer can integrate the update dependencies of an update into an *update procedure* which has the notion of success or failure. This procedure is activated whenever an attempt is made to perform a tuple-level update (i.e. there is one activation of a procedure for each updated tuple). If the procedure fails, the update is rejected and the database remains unchanged; if the procedure succeeds, the update is granted and the database is modified to correctly contain the update. Updates to the database can only be performed by update procedures. Since a procedure has the option of rejecting or granting an update, user updates are translated into update *requests*. Note that, for languages with set-oriented semantics, several update requests may be generated for one user update. For example, the SQL statement "`update R set` $A_1 = f_1, \ldots, A_n = f_n$ `where P;`" is translated into one update request for each tuple that satisfies the selection `P`. We also make the assumption that if no procedure has been specified for an update, then the update will be processed normally by the RDBMS.

Since an update procedure is activated by a user operation, all updates the procedure makes will be either comitted or aborted atomically with the transaction that issued the operation (the *containing* transaction). Furthermore, the transaction can test the result of the procedure activation and react accordingly. These tests can be explicitly specified by the user or inherent in the semantics of the containing transaction, such as in the execution of update procedures to be discussed in the next section.

2.1 Overview of Update Procedures

Each relation and each view has three update procedures associated with it – one for each type of update (i.e. *insert, delete, or update*). Each procedure groups together a set of candidate rules that might apply to an update that activates the procedure. A rule has a condition which tests the database state, followed by a sequence of actions that are tried if the condition is satisfied. These actions either request other updates, perform external i/o which can be used to get information that is needed to complete the update, or actually make physical updates to the database. Note that the activation of one procedure may activate other procedures including itself.

Each update procedure has a structure represented by the following template:

```
op-type rel-name (a1 = v1, ..., an = vn; b1 = w1, ..., bm = wm)
        -> cond1, act1,1, ..., act1,n1.
        -> ...
        -> condq, actq,1, ..., actq,nq.
```

The first line of this template is the procedure head and each line following an -> represents one candidate rule.

The *procedure head* contains the operation type **op-type** and the relation name **rel-name** of the procedure. There is at most one update procedure for each combination of the **op-type** and **rel-name**. The **op-type** is either **insert**, **delete**, or **update** and the relation name **rel-name** identifies the base relation or view of the update.

The procedure head also contains either one or two attribute-parameter lists which provide a mechanism for passing values between a request and its corresponding procedure activation. Values are passed from a request to its activation when the attribute whose value is supplied by the request is mapped to a variable in one of these lists. The first attribute-parameter list, which occurs in all procedures, maps the values of the request's **selection attributes** a_i to the variables $v_i, 1 \leq i \leq qn$. For procedures with operation type **insert**, these values are the attribute values of inserted tuple. For procedures with operation type **update** or **delete**, these values are the existing attribute values of the tuple that was selected for update or deletion. The second attribute-parameter list, which only occurs in procedures of type **update**, maps the values of the request's *update attributes* b_i to the variables $w_i, 1 \leq i \leq m$. These values are the new attribute values for the updated tuple.

The attribute-parameter lists do not need to contain a variable mapping for each attribute in the procedure's relation. Furthermore, an update request that activates a procedure does not need to provide values for all attributes represented in the parameter lists.

If the latter occurs, then there are variables in the parameter lists which do not have values at the beginning of the procedure activation. However, these variables are guaranteed, by procedure safety (see section 2.4), to be mapped to a value as a result of a successful activation. Hence, this provides a mechanism for passing values back to the activating request which is commonly used by requests that are made from within modification procedures (see section 2.3).

The procedure head is followed by a set of candidate rules. Each *candidate rule* consists of a condition $cond_i$ followed by a sequence of actions $act_{i,j}, 1 \leq j \leq n_i$. The conditions are queries on the database state, written in a language similar to domain relational calculus. The actions perform physical updates to the database, activate other procedures by requesting further updates, or perform external i/o.

When a rule is applied, it first executes its condition as a query to the database. If no tuples satisfy the query the condition fails and, consequently, the rule fails. Otherwise, the rule chooses exactly one tuple that satisfies this query and attempts to sequentially execute all of its actions. Since actions can be update requests that depend on the attribute values of the chosen tuple, they may result in a rejection. Hence, not all tuples that satisfy the condition will result in a successful exection of the actions. If there is at least one tuple that does, then the rule chooses exactly one such tuple (the *chosen tuple*), and succeeds. Otherwise it fails.

An update procedure succeeds if any one of its rules succeed. Exactly one of the successful rules is elected as the *solution rule* and only the effects of this rule persist. If no rules satisfy these requirements, the update procedure fails and the database state remains unchanged. The choice of the solution rule is not explicitly stated by the language, and the order in which candidate rules appear in a procedure is irrelevant. If the designer wants control over which rule is chosen, then the conditions of the candidate rules should be mutually exclusive.

The variables that occur in update procedures are analogous to logical variables [LS86] and behave differently from conventional variables (such as those found in C and Fortran). If a variable has a value, the value cannot be overwritten. However, if the action that mapped a value to a variable is undone, then the mapping is also undone and the variable returns to a state of being *uninstantiated.*

The scope of all variables that occur in the procedure head is the entire procedure, and the scope of a variable that only occurs within a candidate rule is limited to the rule in which it occurs. If a variable is passed a value by the activating request, then the variable acts as a constant in the procedure activation; its value cannot be overwritten. On the other hand, any variables that occur only within the scope of a rule or occur in the procedure head but are not passed a value from the activating request may have different values within the scope of each rule; there is no correlation between common variables in different rules. Since only one rule is chosen as the solution rule, each variable in the head of the procedure will have a unique value at the end of the procedure activation.

When a rule is applied, it maintains a *substitution* which provides a unique mapping of variables to values for all variables in the scope of the rule and its procedure. A variable is *instantiated* if there is a mapping for it in the rule's substitution. This substitution is initialized, through the attribute-paramemeter lists of the rule's procedure, to the values supplied by the activating request. It is updated by the evaluation of the rule's condition and the execution of its actions. As will be explicated in the following section, each tuple that satisfies a condition defines a mapping of values to the condition's variables. A rule's

substitution is appended with the mapping defined by its chosen tuple. Note that the chosen tuple's mapping cannot conflict with the rule's substitution (i.e have a different value for any variable that is already mapped to a value in the rule's current substitution). The substitution must also reflect the mappings that are supplied through external i/o and those that result from successful activations of update procedures (section 2.3). The values returned by an update procedure are the values represented by the substitution of its solution rule.

2.2 Conditions

Conditions are queries on the database state that are written in a language that is similar to domain relational calculus (DRC) [Ull88]. DRC provides a convenient notation for indicating which parameter values in the procedure head are substituted for variables in the condition, and which attribute values of tuples that satisfy the condition map to variables used in the actions of the rule. *Instantiation tests*, which are meta-logical predicates (à la Prolog [LS86]), are included for testing if a parameter is supplied in a given activation of a procedure.

As with DRC, our conditions are defined recursively. The *atomic formulas* in this condition sub-language are defined as follows:

literal: `rel-name`$(a_1 = v_1, \ldots, a_k = v_k)$
Evaluates to true if there exists at least one tuple in relation (or view) `rel-name` such that, for every instantiated variable v_i, the value of attribute a_i is equal to the value of v_i. Every uninstantiated variable v_j will be instantiated as a result of the evaluation. The instantiated variables act as selection values and the uninstantiated variables act as either join or return value variables.

comparison: `X` θ `Y`, $\theta \in \{<, \leq, =, \neq, \geq, >\}$
Evaluates to true if the algebraic relation θ holds between symbols `X` and `Y`. `X` and `Y` are either constants or variables.

empty condition: Always evaluates to true. This condition is useful for a rule that is always possible for a given procedure, independent of the database state.

negative-instantiation test: `var`(v_i)
Evaluates to true if the variable v_i, introduced in the procedure head, is not supplied by the update request that activated the current procedure.

positive-instantiation test: `nonvar`(v_i)
Evaluates to true if the variable v_i, introduced in the procedure head, is supplied by the update request that activated the current procedure.

Conditions can be combined, negated, and quantified to form other conditions as follows:

negation: `not C`

Evaluates to true if the sub-condition `C`, which cannot contain any instantiation tests, does not evaluate to true.

conjunction: C_1 `AND ...` `AND` C_n

Evaluates to true if every condition C_i, $1 \leq i \leq n$ evaluates to true.

disjunction: C_1 `OR ...` C_n

Evaluates to true if at least one condition C_i, $1 \leq i \leq n$ evaluates to true.

existential quantification: `exists` v_1 `...` v_n `C`

Evaluates to true if there is at least one mapping of values to variables v_i, $1 \leq i \leq n$ that satisfies the sub-condition `C`. `C` cannot contain any instantiation tests, and there must be at least one occurrence of each variable v_i that is free in `C`.

nesting: (C_i)

Evaluates to true if C_i evaluates to true. Parenthesis are used to alter the default precedence of operators. The unary operators, negation and existential quantification, have the highest precedence and are applied right to left when they occur consecutively. As usual, conjunction has a higher precedence than disjunction.

Occurrences of variables in conditions are distinguished as either *free* or *bound.* All occurrences of variables in an atomic formula `C` are free in `C`. An occurrence of a variable in a condition `C` that is a negation, conjunction, disjunction or nesting is free or bound in `C` if it is free or bound in the subcondition in which it occurs. All free occurrences of variables v_i, $1 \leq i \leq n$ in `C` are bound in the existential quantification `exists` v_1 `...` v_n `C`; all other occurrences of variables in existential quantifications are free or bound if they are free or bound in the subconditions in which they occur.

In DRC, variables that occur free in a condition define the relation that corresponds to the condition. In our language, only the instantiated-free variables define the relation that correspond to the condition. A free variable is not an instantiated-free variable if it only occurs in negative instantiation tests in the condition.

Condition Safety

There are restrictions that must be imposed on conditions to insure that they are *domain-independent.* A condition is domain-independent if, for any given instantiation of the variables from the procedure head, the relation generated by the condition depends only on the constants in the condition and the domains of the relations named in the condition. For this reason, universal quantification is not included in our condition sublanguage. This does not reduce the expressiveness of the conditions since universal quantification can be expressed with negation and existential quantification. Domain-independence is guaranteed by insuring that conditions are *safe.*

A condition of a rule is safe if all instantiated-free variables in the condition are *limited*, and any variable that occurs bound in an existential quantification **exists** v_1 ... v_n C is limited in the sub-condition C. [2] Each condition can be considered as a conjunction of one or more sub-conditions. For the following discussion, let C = C_1 **AND** ... **AND** C_n. A variable v_i is limited in C if one of the sub-conditions in which it occurs is of the following type:

1. positive instantiation test, nonvar(v_i). If a variable satisfies this test, the value of the variable is supplied by the originating request and is therefore a constant with respect to the condition.
2. literal, rel-name($a_1 = x_1$, ..., $a_k = x_k$) where v_i is one of the x_i. Since the underlying database is finite, there are only a finite number of possible values for the variable.
3. equality comparison, v_i = X or X = v_i where X is a constant or limited.
4. a disjunction K_1 **OR** ... **OR** K_q in which v_i is limited in all K_j. Hence, disjunctions are only allowed when they specify either a union or a selection condition for a given set of variables. Any variable that occurs in a disjunction that is not otherwise limited by the enclosing condition, must be limited in all sub-conditions of the disjunction.
5. an existential quantification **exists** v_1 ... v_n K, in which $v_i \neq v_j$, $1 \leq j \leq n$ and v_i is limited in K.

Any instantiated-free variable that appears in either a negation test or a negative-instantiation test, which are sources of domain-dependence, must also occur in one of the above types of sub-conditions that will limit the domain of the variable. The safeness of variables that are not instantiated-free is guaranteed by procedure safety, which is discussed later.

2.3 Actions

There are three types of actions: doit-actions, req-actions, and i/o-actions. Doit-actions perform the physical updates for the request that activated the procedure, while req-actions request other updates from within the procedures. I/o actions perform i/o operations that are external to the database and can be used to get information that is needed to complete the update or to notify the user of the actions taken by the database.

Doit-actions have one of the following forms:

[2] The discussion of safe conditions here is similar to that in [Ull88] with a few minor exceptions. We allow the meta-logical type predicates to limit variables. We also relax the safety conditions of disjunctions and incorporate these conditions into the definition of limited.

```
ins rel-name(a1 = e1, ..., an = en),
del rel-name(a1 = e1, ..., an = en),
upd rel-name(a1 = e1, ..., an = en; b1 = f1, ..., bm = fm).
```

The expressions e_i and $f_i, 1 \leq i \leq n$, are evaluated using the current substitution of the action's rule. All variables that occur in these expressions must be instantiated. As with update procedures, there are two types of attribute-parameter lists. However, these lists serve to provide values from the variables to the attributes.

The first attribute-parameter list occurs in every doit-action and the second one only occurs in doit-actions of type upd. Not all attributes of relation `rel-name` must be present in each attribute-parameter list. However, for actions of type `ins`, the value for any attribute not specified in the attribute-parameter list is initialized to `NULL`. For actions of type `del` and upd, the set of attributes $a_1, \ldots, a_n$ in the first attribute-parameter list must contain the attributes in the key of the relation `rel-name`. For upd actions, the second attribute-parameter list indicates the update values for the attributes $b_1, \ldots, b_m$.

Req-actions have a form that is similar to the procedure heads:

```
insert rel-name(a1 = e1, ..., an = en),
delete rel-name(a1 = e1, ..., an = en),
update rel-name(a1 = e1, ..., an = en; b1 = f1, ..., bm = fm).
```

The expressions $e_i, 1 \leq i \leq n$, and $f_i, 1 \leq i \leq m$, are either uninstantiated variables or arithmetic expressions whose variables are instantiated. Those expressions that are uninstantiated will be mapped to a value if the request is granted and this mapping will be added to the substitution of this action's rule. If the expression e_i (f_j) is an uninstantiated variable, then it will be instantiated to the value of the variable that corresponds to a_i (b_j) in the procedure that is activated by the request. Note that update procedures may be recursive since an action may request an update that activates the action's update procedure.

The *i/o-actions* `read` and `write` provide interaction between procedures and external sources. A write must follow an operation that substitutes a value for the write's variable. Hence, the variable must occur as either [3]

- a limited variable in the condition $cond_i$ or
- a parameter in a read or req-action $act_{i,k}, k < j$.

On the other hand, a read cannot follow any operation that substitutes a value for the read's variable. Hence, the variable of the read action $acti, j$ cannot occur as

[3] If the variable occurs in the procedure head, its value must be supplied by the user or obtained from the database. In either case it will be limited by the condition.

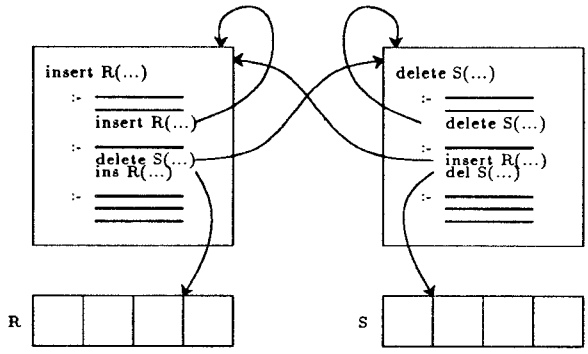


Figure 1: Encapsulation of Updates

- a limited variable in the condition $\texttt{cond}_i$,
- a parameter in any read or req-action $\texttt{act}_{i,k}, k < j$, or
- a parameter in the procedure head, unless it occurs as an unlimited variable in a negative instantiation test.

All actions have the notion of success or failure. Since req-actions request other updates, they succeed only if the request is granted. Doit-actions and write i/o-actions always succeed. [4] However, since read actions depend on receiving some input from either an interactive terminal or a file, they may not always succeed. If the request is being processed by an interactive process and the user does not respond within some fixed amount of time the read will fail. The read also fails if it is being processed by a non-interactive process unless a non-empty input file was specified when the process was initiated.

Since doit-actions make the physical update to the database, they are only allowed from an update procedure that has the same operation type and relation as the doit-action. Furthermore, they are not allowed in update procedures for views. Update requests for a view are performed by requesting updates on the base relations from which the view is derived. These restrictions encapsulate all update access to a relation in the relation's three update procedures. Figure 1 shows that the the doit-action `ins R(....)`, which is defined only if `R` is a base relation, is only allowed in the procedure `insert R(...)`, but procedure `insert R(...)` can be activated by a req-action from another procedure.

When an action is performed, its effects are only visible to actions that subsequently follow it in the candidate rule unless its candidate rule succeeds and is chosen as the solution

[4] Note that these actions may fail if there is a database or system failure, however, we do not consider these kinds of failures here.

rule. If this is the case, the effects of all the actions of the rule become the effects of the request and are visible only where the effects of the request are visible. If the request is from a user update, then visibility of the request's effects is governed by the containing transaction. If the request is from another update procedure, then this visibility is governed by the requesting update procedure. Section 4 describes two execution strategies which give executional semantics for the evaluation of an update procedure.

2.4 Procedure Safety

There are constraints that must be imposed on the update procedures to insure that any given execution is domain-independent. There are two sources of domain-dependence in these procedures that can be eliminated by applying the constraints for safe rules and safe domain-relational-calculus formulas as described in [Ull88]. We define the concept of "safe" procedures that adapts these two concepts to our update procedures. A safe procedure is one in which each candidate rule is safe with respect to the procedure head and all conditions of the candidate rules are safe (as defined in section 2.2).

Notice, for the purpose of defining safety, a procedure with head h and candidate rules $b_i, 1 \leq i \leq n$, is similar to a set of Datalog rules $d_i, 1 \leq i \leq n$, such that h is the head of each d_i and b_i is the body of d_i. The safe rule requirements apply to each candidate rule with respect to the procedure head. Every variable in the procedure head must be limited in each candidate rule. A variable is limited in a rule if it is limited in the rule's condition or it occurs either in a `req` or a `read` action. Furthermore, any variable in a `mod` action $act_{i,j}$ must be limited by the condition or by some preceding action $act_{i,k}, k < j$.

3 Examples

Example 1 View Materialization

Assume that the view `v(x,y,z)` is defined as the join of `r(x,y)` with `s(y,z)`. If the system maintains a materialized version of `v` (i.e. the database administrator defines `v` as a base relation, but `v` is modified only by updates to `r` and `s`), then update procedures can be used to specify the incremental maintenance of `v`. The following update procedures perform incremental maintenance for insertions to `v` that result from insertions to `r` and `s`. We show only the tuple insertion rule for `r`; the rule for `s` is similar.

```
insert v(x=v1, y=v2, z=v3)
-> nonvar(v1) and nonvar(v2) and nonvar(v3) and
   not v(x=v1, y=v2, z=v3),
   ins v(x=v1, y=v2, z=v3).
```

```
insert r(x=v1, y=v2)
-> r(x=v1, y=v2) and
   not exists v3 (s(y=v2, z=v3) and not v(x=v1, y=v2, z=v3)).
-> r(x=v1, y=v2) and
   v3 (s(y=v2, z=v3) and not v(x=v1, y=v2, z=v3)),
   insert v(x=v1, y=v2, z=v3),
   insert r(x=v1, y=v2).
-> not r(x=v1, y=v2),
   ins r(x=v1, y=v2),
   insert r(x=v1, y=v2).
```

The insertion procedure for v simply enforces a set-semantics (i.e. no duplicates). The insertion procedure for r recursively triggers insertions into v until v is consistent with r. Note that the conditions for the rules of this procedure are mutually exclusive. If there were a fixed order for selecting qualified rules, the designer may have been tempted to use this order to imply semantics about the conditions. For example, suppose rules are tried in the order they appear in the procedure. Then, she may have been tempted to exclude the existential clause from the second rule's condition and to not include any condition for the third rule.

Note that it is possible to generate correct update procedures for incrementally maintaining views. Incremental maintenance is a a fundamental design goal of ADMS [Rou91], which performs incremental maintenance for all SPJ views defined. Recently, [CW91] specified how Starburst production rules can be generated to support incremental view maintenance.

■

Example 2 View Update - Application Independent

Suppose that an application requires the ability to specify updates for the view v defined in the previous example. Then v is defined as a base relation with the additional integrity constraint, `v(x,y,z) =r(x,y) join s(y,z)`. Update procedures to maintain this integrity constraint must be specified. The following is one such insertion procedure for v, where the tuple insertion rules on r and s are specified above.

```
insert v(x=v1, y=v2, z=v3)
-> not v(x=v1, y=v2, z=v3),
   ins v(x=v1, y=v2, z=v3),
   insert r(x=v1, y=v2),
   insert s(y=v2, z=v3).
```

The tuple procedure for v makes the insertion into v and requests insertions into r and s; these, in turn, recursively request any additional needed insertions into v as above. This

is another example where the update procedure can be generated from the view definition. The view contains all attributes from its deriving relations and hence this update is well-defined, independent of the application semantics of v. ∎

Example 3 View Update - Application Dependent

Recall the insurance database from the section 1 that consisted of the base relations `insured(ins-no, patient)` and `accounts(ins-no, payer)` and the view `dependent(payer, patient)` in which dependent patients are those persons whose insurance is covered by some other person. The following update procedures support the deletion, insertion and update through the view defined above for one given set of application specific rules.

The only restrictions that exist for this application are:

- a patient can only be inserted as a dependent of a payer that has exactly one policy. (This restriction can be lifted by accepting a dependent to be inserted under any one of the insurances paid by the payer, or by adding information to the base relations that indicates the primary insurance policy for a given person.)
- when a dependent tuple is deleted, then the insured patient is no longer covered by the payer's policy. The account relation is not affected by updates to the dependent view.

```
delete dependent(payer=p, patient=q)
/* delete q as a dependent of p  */
-> nonvar(p) and nonvar(q) and p<>q and
   accounts(ins-no=s, payer=p) and insured(ins-no=s, patient=q),
   delete insured(ins-no=s, patient=q),
   delete dependent(payer=p, patient=q).
-> nonvar(p) and nonvar(q) and
   not exists s (accounts(ins-no=s, payer=p) and insured(ins-no=s, patient=q)
   and p<>q).
/* delete q as a dependent of any p */
-> var(p) and nonvar(q) and
   insured(ins-no=s, patient=q) and account(ins-no=s, payer=p) and p<>q,
   delete insured(ins-no=s, patient=q),
   delete dependent(patient=q).
-> var(p) and nonvar(q) and
   not exists s (insured(ins-no=s, patient=q) and account(ins-no=s, payer=p) and
   p<>q).

insert dependent(payer=p, patient=q)
/* q is inserted as a dependent of p on p's one and only account */
-> nonvar(p) and nonvar(q) and p<>q and
```

```
   account(ins-no=s1, payer=p) and
   not exists s2 (account(ins-no=s2, payer=p) and s1<>s2) and
   not insured(ins-no=s1, patient=q),
   insert insured(ins-no=s1, patient=q).

update dependent(payer=p1, patient=q; payer=p2)
/* q is removed as a dependent from (all) p1's account(s) and
   made a dependent on p2's one and only account */
-> nonvar(p1) and nonvar(p2) and nonvar(q) and account(payer=p2),
   delete dependent(payer=p1, patient=q),
   insert dependent(payer=p2, patient=q).

/* q is removed as a dependent from all accounts and
   made a dependent on p2's one and only account */
-> var(p1) and nonvar(p2) and nonvar(q) and account(payer=p2),
   delete dependent(patient=q)
   insert dependent(payer=p2, patient=q).
```

■

Example 4 Data Evolution

Suppose an IRS database records information about taxpayers and dependents. Information about independent taxpayers is recorded in the relation `taxpayer(payer-ss#, payer-name, addr, #dep)` with key `payer-ss#` and the dependents of all independent taxpayers is recorded in the relation `dependent(payer-ss#, dep-ss#, dep-name)` with key `dep-ss#`. The following rules govern the evolution of data in this database:

- a person cannot simultaneously be an independent taxpayer and a dependent of an independent taxpayer.
- no person can claim themselves as a dependent
- A person is never deleted from the database. However, dependents can evolve into independent taxpayers.

The following set of update procedures maintains these rules.

```
insert taxpayer(payer-ss#=s, payer-name=n, addr=a)
-> nonvar(s) and nonvar(n) and nonvar(a) and
   not taxpayer(payer-ss#=s) and not dependent(dep-ss#=s),
   ins taxpayer(payer-ss#=s, payer-name=n, addr=a, #dep=0).
-> nonvar(s) and nonvar(n) and nonvar(a) and
   not taxpayer(payer-ss#=s) and dependent(payer-ss#=p, dep-ss#=s),
   ins taxpayer(payer-ss#=s, payer-name=n, addr=a, #dep=0),
```

```
   delete dependent(dep-ss#=s),
   update taxpayer(payer-ss#=p, #dep=nd; #dep=nd - 1).

insert dependent(payer-ss#=p, dep-ss#=d, dep-name=n)
-> nonvar(p) and nonvar(d) and nonvar(n) and
   not taxpayer(payer-ss#=d),
   update taxpayer(payer-ss#=p, #dep=nd; #dep=nd+1),
   ins dependent(payer-ss#=p, dep-ss#=d, dep-name=n).

delete dependent(dep-ss#=d)
-> nonvar(d) and dependent(payer-ss#=p, dep-ss=d, dep-name=n)
   and taxpayer(payer-ss#=p, addr=a) and not taxpayer(payer-ss#=d),
   del dependent(dep-ss#=d),
   insert taxpayer(payer-ss#=d, payer-name=n, addr=a, #dep=0).
-> nonvar(d) and taxpayer(payer-ss#=d),
   del dependent(dep-ss#=d).
-> nonvar(d) and not dependent(dep-ss#=d) and
   taxpayer(payer-ss#=d).
```

The procedure for updating taxpayers is left as an exercise to the reader. It should ensure that the taxpayer being updated is an independent taxpayer.

■

Example 5 Non-full Functional Dependencies

It is often undesirable to normalize a relation, in which case functional dependencies that have a determinant which is not a candidate key of the relation cannot be enforced by key constraints. For example, the relation `shipments(s#, sname, saddr, p#, qty)` has the functional dependencies `s#` $\rightarrow$ (`sname`, `saddr`), and (`s#`,`p#`) $\rightarrow$ `qty`. Both `s#` and (`s#`, `p#`) are required as keys to enforce these functional dependencies but cannot coexist as such. The following set of procedures ensures that updates to relations do not violate the functional dependencies despite the fact that the relation is not properly normalized. Note that, in several cases, the actions taken to enforce the functional dependencies are one of several possible solutions and the choice depends on the semantics of the data.

```
delete shipments(s#=s, p#=p)
-> nonvar(s) and nonvar(p) and shipments(s#=s, p#=p),
   del shipments(s#=s, p#=p).
-> nonvar(s) and nonvar(p) and not shipments(s#=s, p#=p).
```

The first rule of this procedure assumes that the key (`s#`,`p#`) uniquely identifies a tuple. The second rule allows the deletion of non-existing tuples.

```
insert shipments(s#=s, sname=n, saddr=a, p#=p, qty=q)
-> nonvar(s) and nonvar(n) and nonvar(a) and nonvar(p) and nonvar(q) and
   not shipments(s#=s),
   ins shipments(s#=s, sname=n, saddr=a, p#=p, qty=q).
-> nonvar(s) and nonvar(p) and nonvar(q) and
   not shipments(s#=s, p#=p) and shipments(s#=s, sname=n, and saddr=a),
   ins shipments(s#=s, sname=n, saddr=a, p#=p, qty=q).
```

This procedure maintains the functional dependencies during insertions to shipments. The first rule handles the case when the inserted shipment is the first one for the specified supplier. In the second rule, the specified supplier is the supplier for existing shipments, but does not supply the specified part for any of these shipments. In this case, if the specified (sname,saddr) are inconsistent with those in the database for the specified supplier, then the insertion fails. Otherwise, it succeeds. Note that the procedure assumes that the functional dependencies are enforced at the outset.

```
update shipments(s#=s1, p#=p1; s#=s2, sname=n2, saddr=a2, p#=p2, qty=q2)

/* Update p#; s# remains the same */
-> nonvar(s1) and nonvar(p1) and nonvar(p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   p1<>p2 and s1=s2 and not shipments(s#=s2,p#=p2),
   upd shipments(s#=s1, p#=p1; p#=p2, qty=q2),
   update shipments(s#=s2; sname=n2, saddr=a2).

/* Update s# to a new supplier; p# may be updated as well;
   supplier name and address must be given */
-> nonvar(s1) and nonvar(p1) and nonvar(s2) and s1<>s2 and
   nonvar(n2) and nonvar(a2) and
   ((nonvar(p2) and p1<>p2) or p1=p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   not shipments(s#=s2),
   upd shipments(s#=s1, p#=p1; s#=s2, p#=p2, sname=n2, saddr=a2, qty=q2).

/* Update s# to existing supplier; p# may updated as well; */
-> nonvar(s1) and nonvar(p1) and nonvar(s2) and s1<>s2 and
   ((nonvar(p2) and p1<>p2) or p1=p2) and
   (nonvar(q2) or shipments(s#=s1,p#=p1,qty=q2)) and
   shipment(s#=s2, sname=x, saddr=y),
   upd shipments(s#=s1, p#=p1; s#=s2, p#=p2, sname=x, saddr=y, qty=q2),
   update shipments(s#=s2; sname=n2, saddr=a2).

/* Update of sname and saddr */
-> nonvar(s1) and s1=s2 and
```

```
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2))) and
   shipments(s#=s1, p#=p1, sname=x, saddr=y, qty=q2) and
   (nonvar(n2) or n2=x) and (nonvar(a2) or a2=y) and (x<>n2 or y<>a2),
   upd shipments(s#=s1, p#=p1; sname=n2, saddr=a2),
   update shipments(s#=s1; sname=n2, saddr=a2).

/* Termination rules */
-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2))) and
   nonvar(n2) and var(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists x (shipments(s#=s1, sname=x) and x<>n2).

-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2))) and
   var(n2) and nonvar(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists y (shipments(s#=s1, saddr=y) and y<>a2).

-> nonvar(s1) and s1=s2 and
   ((nonvar(p1) and p1=p2) or (var(p1) and var(p2) and var(q2))) and
   nonvar(n2) and nonvar(a2) and
   shipments(s#=s1, p#=p1, sname=n2, saddr=a2, qty=q2) and
   not exists x y (shipments(s#=s1, sname=x, saddr=y) and (x<>n2 or y<>a2)).
```

This procedure maintains the functional dependencies during insertions to shipments. The rules isolate updates to different fields; one rule modifies one field, and requests all other updates recursively. The first rule performs all updates to part numbers (p#) when the supplier number (s#) remains the same. Any further updates of supplier name and addresses are handled by a recursive request. The second and third rules handle all cases where the supplier is updated. For the second rule, the update must be updating the supplier number to a number that is not already in the database. If this is the case, then the supplier name and address must also be given, and the execution terminates. The third rule updates the supplier number to a number for an existing supplier. A shipment tuple satisfying the selection criterion is updated with the new supplier number given the name and address of another shipment in the database with the new supplier number. Any further updates of supplier name and address are handled by a recursive request and performed by the fourth rule. To maintain the functional dependencies, all shipments with the same supplier number must be updated with the new supplier name and address, and this is also handled by a recursive request to the same procedure. The second, fifth, sixth, and seventh rules terminate execution of the procedure. The fifth, sixth, and seventh rules have empty action

clauses, and serve just to successfully terminate the procedure. ∎

Example 6 Dependency Preservation

It is often impossible both to preserve intra-relational dependencies and to obtain Boyce-Codd normal form (BCNF) in a relational scheme. The well-known SJT example from [Dat86] illustrates this problem. The database designer wants to model class enrollment with the relation `enroll(student,teacher,subject)`. However, there are two restrictions about the class enrollment that must be observed. First, each student takes a given subject from only one teacher; this is represented by the functional dependency `(student,subject)` → `teacher`. Second, each teacher teaches only one subject, but there may be several teachers of the same subject; this is represented by the functional dependency `teacher` → `subject`. So the original relation, `enroll(student,teacher,subject)`, is not in BCNF since `teacher` → `subject` and `teacher` is not a candidate key of `enroll`. Hence, the original relation is replaced with the two projections `roster(student,teacher)` and `courses(teacher,subject)`, in which the original intra-relational functional dependency `(student,subject)` → `teacher` is no longer preserved. If the two projections are updated independently, they may violate this dependency. However, this functional dependency can be expressed as an inter-relational constraint by specifying update procedures for the insert and update requests of the two projections. The following are the update procedures for `insert` requests:

```
insert roster(student=x, teacher=y)
-> nonvar(x) and nonvar(y) and
   not roster(student=x, teacher=y)
   and not exists  y1 z (roster(student=x, teacher=y1) and
      courses(teacher=y1, subject=z) and
      courses(teacher=y, subject=z) and
       y1 <> y)),
   ins roster(student=x, teacher=y).
-> roster(student=x, teacher=y).

insert courses(teacher=y, subject=z)
-> nonvar(x) and nonvar(y) and
   not courses(teacher=y) and
   not exists x  y1 (roster(student=x, teacher=y1) and
       courses(teacher=y1, subject=z) and
       roster(student=x, teacher=y) and
       y1 <> y)),
   ins courses(teacher=y, subject=z).
-> courses(teacher=y, subject=z).
```

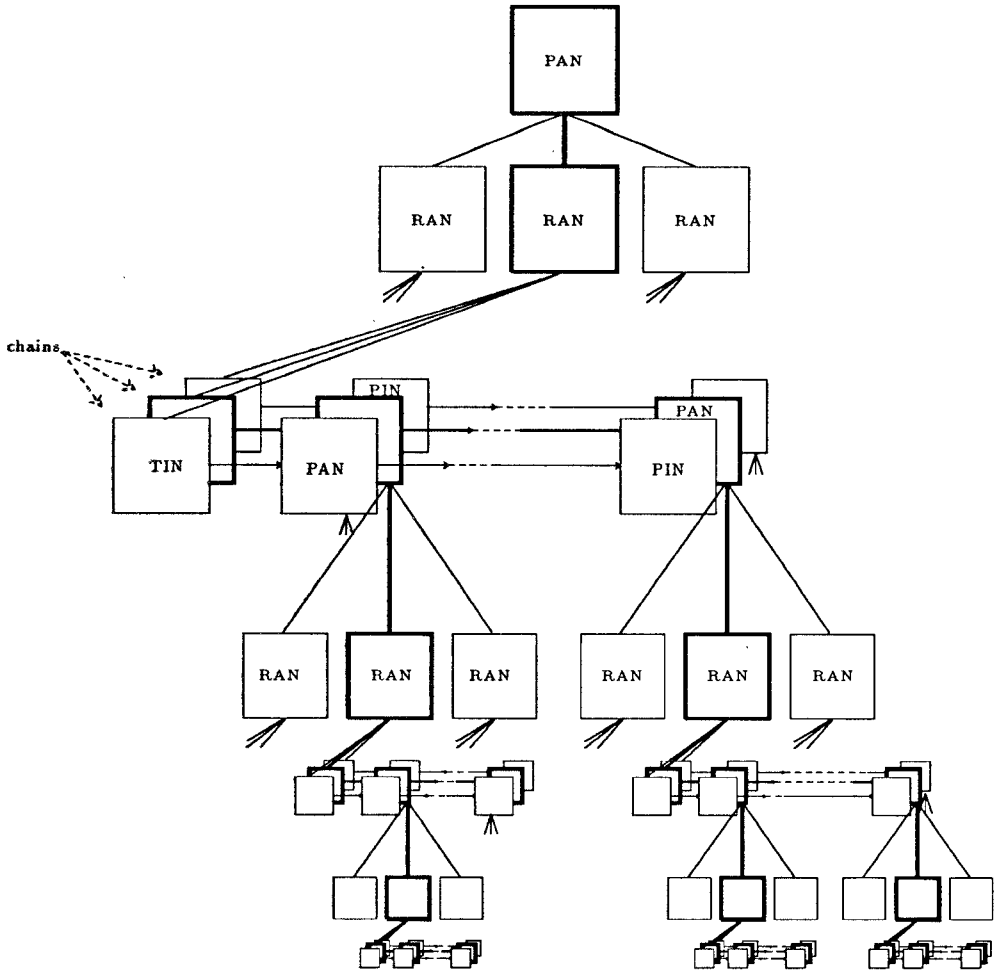


Figure 2: Execution Pyramid

Notice that the last rule in each procedure enforces a set-semantics: attempts to insert a duplicate tuple returns true without modifying the database. ∎

4 Execution Strategies

In this section we first define *the update dependency execution pyramid* (Figure 2) which is a model of execution for update procedures. We use this model to describe two control strategies for executing the procedures.

Pyramids are related to AND/OR trees [Nil80]. They contain nodes that are linked together in parent-child hierarchies and ordered-lists called *chains*. A parent-child hierarchy represents a disjunction between siblings; the parent is analogous to an OR-node. Chains

represent a conjunction between the nodes of the chain in which the nodes are evaluted in a pre-defined sequential order; the first node in the chain is similar to an AND-node. It is this sequential execution of the nodes in a chain that distinguishes pyramids from AND/OR trees. The pictorial representation of the chains in the parent-child hierarchy abstractly looks like a pyramid.

A pyramid contains four types of nodes:

procedure activation nodes (PANs)
: represent activations of update procedures from user transactions or from rule applications.

rule application nodes (RANs)
: represent applications of candidate rules from procedures.

tuple instance nodes (TINs)
: represent tuples in the database that satisfy conditions.

primitive nodes (PINs)
: represent doit-actions and i/o-actions.

The root of a pyramid is a PAN, representing the procedure activated for the update requested by a user transaction. Each PAN has one or more children RAN nodes, representing the disjunction between the candidate rules of the PAN. All non-root PAN nodes occur in *chains* and represent req-actions.

A RAN is the condition choice point for a rule application. It has one child TIN for each tuple that satisfies the rule's condition. If the condition of a rule cannot be satisfied, then the RAN will not have any children. If the condition of a rule is the literal "true", the RAN will have exactly one TIN child which represents the empty tuple. All the TIN nodes of a RAN are in a disjunction, representing the tuple-oriented semantics of the rules.

A chain links a sequence of PANs and PINs to a TIN node. The first node of a chain is always a TIN node, and TIN nodes only occur as the first node of a chain. The chain's remaining nodes represent instances of the rule's sequence of actions.

A leaf in a pyramid is a node that has no children. TINs and PINs are always leaves, even though they may be linked in a chain. RANs whose condition cannot be satisfied by the database are also leaves. Since update procedures must always have at least one candidate rule, a PAN will always have at least one child and will never be a leaf.

The execution pyramid for an activation may be infinite since update procedures can be recursive. At first it might seem that the search space could be reduced without affecting the semantics by preventing multiple requests of the same update with the same set of parameters. This is true only if the database state did not change between two identical

requests. But since the database state may change between requests, and the success of a candidate rule depends on this state, we cannot perform such optimizations.

The successful executions of an activation are sub-pyramids, called solution pyramids, of the activation's corresponding execution pyramid. Solution pyramids have the following properties:

- each PAN node has exactly one child RAN,
- each RAN node has exactly one child TIN,
- the substitutions for each node in a chain are consistent,
- all leaf nodes of the pyramid are either PINs or TINs.

Several solution pyramids are show in boldface in Figure 2.

The execution of an update procedure activation is equivalent to searching the activation's execution pyramid for a solution pyramid. Note that there may be several solution pyramids for a given activation of an update procedure. Only one of these is chosen by the execution strategy as the solution pyramid. The sequence of actions and queries that are performed as a result of the execution corresponds to a left-to-right traversal of the leaf nodes of the chosen solution pyramid. TINs represent database reads, and PINs represent either database updates or i/o-actions. A TIN never reflects database updates that correspond to leaf nodes that follow it in this traversal, but it may reflect database updates that correspond to leaf nodes that precede it.

Figure 3 shows a subset of a database and the corresponding solution pyramid for the database request `delete(dependent(payer=''Joe'', patient=''Bobbie''))` when the update procedure from example 3 has been defined. Note that the two TIN nodes are leaves, and that the update procedure `delete(dependent(...))` is activated from both a user transaction and from within a candidate rule.

In the remainder of this section, two tentative A *tentative* control strategy is one in which a rule is selected (either arbitrarily or using some heuristics) and applied, but provisions are made to return later to the point immediately before the rule was applied to try some other rule [Nil80]. control strategies for searching an execution pyramid are explored. Both strategies involve dynamically building the solution pyramids, sometimes searching paths in the execution pyramid that must be undone. The first strategy performs depth-first search, using a single process, and backtracks when it discovers that it is building a non-solution pyramid. The second strategy performs a concurrent search for solution pyramids, exploiting parallelism among all alternative child links.

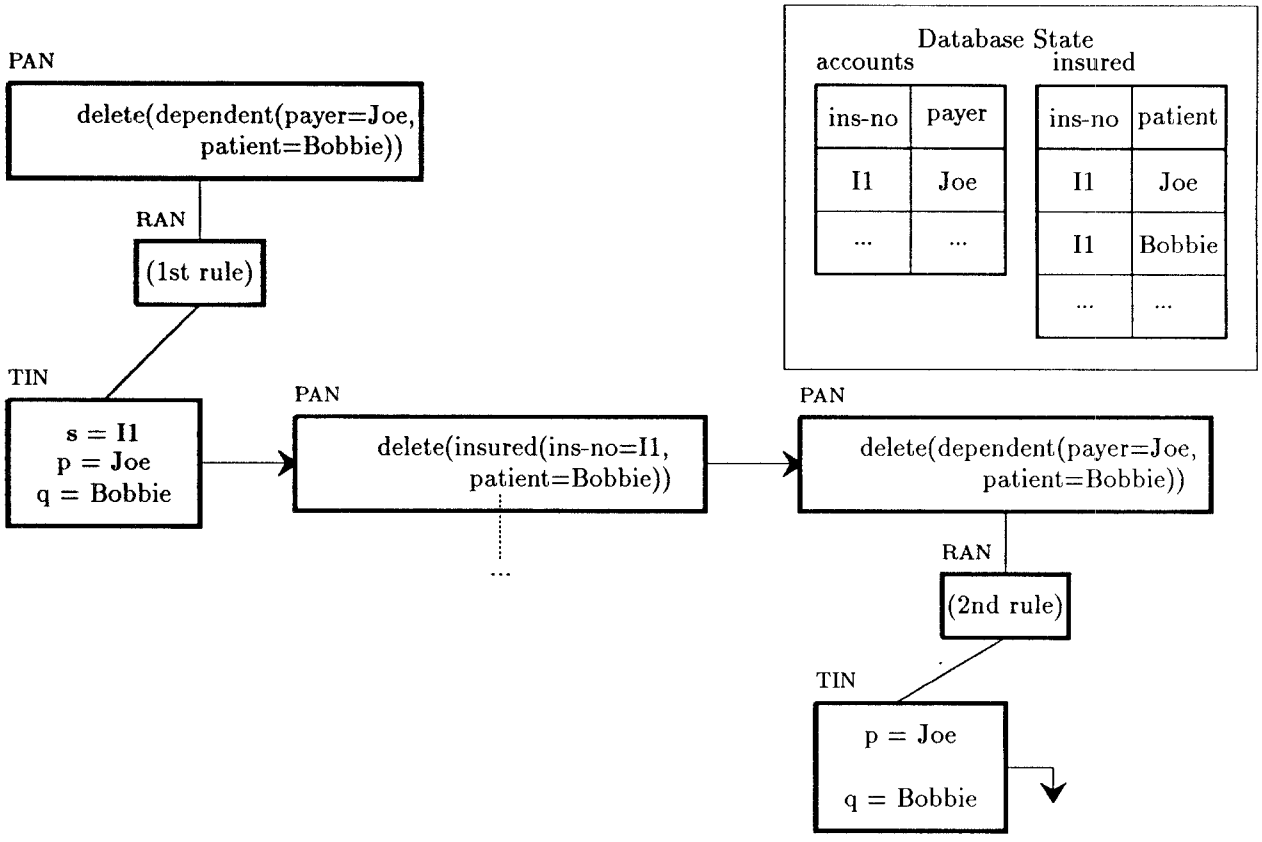


Figure 3: Sample Solution Pyramid

4.1 A Depth-First Search Strategy

Like all depth-first search (DFS) techniques, the DFS of a pyramid (PDFS) must maintain a stack of choice points and return control to the most recent choice point whenever a node fails. There are two types of choice points for pyramids: the selection of a RAN for a PAN and the selection of a TIN and its associated chain for a RAN. The selection of a RAN represents the selection of a rule to apply for a given activation of a procedure. The selection of a TIN and its associated chain represents the selection of a substitution for the variables in the rule's condition.

What makes PDFS unique is the existence of chains in the pyramid. A PDFS of a chain must maintain the sequential order between the nodes in the chain. Since the variable substitutions for nodes in the same chain must be consistent in a solution pyramid, a PDFS must also maintain the left-to-right dependency of nodes in the chain on the current variable substitution and the database state. When a PDFS backtracks, it must undo any variable mappings and database updates that occured after the backtrack point.

PDFS begins at the root PAN. At a PAN, PDFS chooses a RAN to investigate, initializing the RAN's variable substitution according to the attribute values specified by the activating request.

A PDFS of a RAN involves choosing one of the RAN's TINs. TINs represent mappings of values to variables. Only a subset of a RAN's TINs are applicable for a given instance of the rule application since several of the TIN's mappings may not be consistent with the current substitution. So, PDFS chooses one TIN that represents a mapping which is consistent with the RAN's current substitution.

A PDFS must then search for a successful execution of the TIN's chain. To search the chain, the sub-pyramids rooted at each node in the chain are searched in the order in which they occur. After each execution of a node, the RAN's substitution S is updated to reflect any variable mappings obtained by the node. If the node is a PIN, all external reads must be reflected in the RAN's substitution. If the node is a PAN, then all uninstantiated variables in activating request will have a value upon the successful completion of the PAN.

When the search reaches the end of a chain it returns control to the RAN that is the parent of the chain's TIN. The RAN, in turn, returns control to its parent PAN, passing along appropriate return values for the variables in the PAN's modification request. If this PAN is the member of a chain, PDFS updates the substitution of the chain's RAN with the appropriate return values, and continues to process the next node in the chain. If the PAN is the root, then the search terminates successfully.

Since pyramids can potentially be infinite structures, PDFS may get stuck searching an infinite alternative. This can be prevented by imposing a bound for the maximum height

of the search pyramid. Whenever this height is reached, PDFS assumes that the current node fails and backtracks to the most recent choice-point.

A PAN fails when all of its RANs fail, reflecting that a procedure activation fails if none of the procedure's rules can be successfully applied given the input attribute values and the database state. A RAN fails when it is impossible to successfully complete at least one of the TIN's chains, reflecting that there are no tuples that satisfy the rule's condition, are consistent with the rule's substitution, and for which a successful execution of the rule's actions can be found. A chain cannot be successfully completed when there is not a successful PDFS of the chain under the rule's substitution appended with the substitution implied by the chain's TIN. Since PINs fail, then a chain fails when one of its PANs fails.

If a failure is followed to the node where backtracking must occur, this node is always a RAN for which no TINs are consistent with the current variable substitution. When this failure happens, the search must find another substitution by chosing a different TIN for a previously searched RAN or a different RAN for a previously searched PAN. In a PDFS strategy, this is done by backtracking to the most recent such choice point. When this occurs, all database updates and mappings of values to variables that occured after the choice point must be undone.

To enable backtracking, a state for all choice points is maintained in a stack. Each state is a triple $< N, S, T >$ defined as follows:

N the node in the pyramid where the execution should resume.

S the substitution of values for variables before N is applied. This substitution is affected by the external activation, TINs, external reads, and the activation of other update procedures. We assume the proper handling of scope as is typical in all procedure oriented languages.

T the timestamp assigned by the database when the state enters the stack.

Assume that SUB is the current substitution and TS is the current timestamp assigned by the database. Whenever a PAN is visited, the search pushes the state $< RAN, SUB, TS >$ onto this stack for each of the PAN's RANs. Similarly, whenever a RAN is visited, the search pushes the state $< TIN, SUB, TS >$ onto the stack for each RAN's TINs that is consistent with SUB.

When the search encounters a RAN that does not have any TINs that are consistent with the current substitution, it pops the top node $< N, S, T >$ from the choice-stack. It resets the current substitution to S, the current node to N, and rolls the database back to the timestamp T. Concurrency control for PDFS is discussed in section 5.

4.2 Concurrent Search Strategies

In this section we describe a concurrent search strategy for pyramids, PCS. Several research efforts in logic programming utilize concurrent search to improve the response time of a query [Sha87, KKM83]. Other efforts [Wol88, WS88, CW89, Don89, WO90, GST90] study Datalog [Ull88, Chapter 4] parallelization, which involves bottom-up evaluation of Datalog queries. However, the paradigms investigated in these efforts do not consider the behavior of updates in the concurrently searched substructures. Our main interest in describing a concurrent search strategy for pyramids is to investigate the issues involved in supporting concurrent search in a multiuser database system. In such systems, concurrency control must be provided for any set of transactions that are simultaneously executing. Concurrency control issues are discussed later in section 5.

Concurrent search can be applied only to a restricted set of update procedures. I/o-actions implicitly interface with a sequential medium (either a file or a terminal) and cannot be executed in parallel without confusion. Hence, concurrent search strategies can only be applied to procedures that do not activate i/o-actions. Such procedures cannot contain i/o-actions in their rules, and they can only contain modifications that do not activate i/o-actions.

A search strategy can employ concurrency at any node whose children represent independent parts of the structure. In pyramids, every parent-child hierarcy represents a disjunction in which the siblings are not dependent on each other. Hence, there are two dimensions of concurrency corresponding to the two types of parent-child relationships:

rule concurrency exploits the disjunctive relationship between the sibling RANs of a PAN, indicating that a search strategy can simultaneously (but independently) try to apply all the rules of an update procedure to a given activation.

tuple concurrency exploits the disjunctive relationship between the sibling TINs of a RAN, indicating that, for a given application of a rule, a search strategy can simultaneously try to apply the actions of the rule to each tuple that satisfies the rule's condition.

It is no coincidence that these dimensions of concurrency are identical to the choice-points for PDFS since a concurrent search that exploits both forms of concurrency is equivalent to a breadth first search that uses multiple processes.

As with PDFS, a concurrent search strategy for pyramids has some unique requirements. Since pyramids contain actions that update the database (which is shared by all processes), a concurrent search strategy must ensure that the database updates performed by concurrent processes are mutually exclusive. For the remainder of this section, we assume that processes

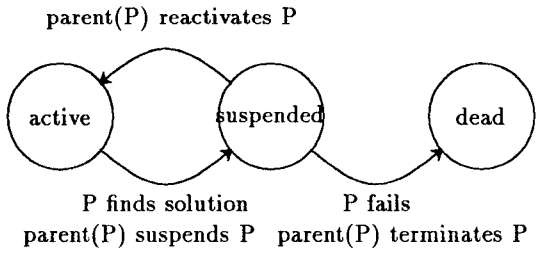


Figure 4: Execution States of Process P

that are concurrently searching subpyramids of disjunctive sibling nodes do not see each other's updates and a parent only sees the updates of its chosen successful child. Mechanisms for obtaining this mutual exclusion are discussed later in subsection 4.2.1.

A concurrent search strategy must also maintain the left-to-right dependencies between the nodes in a chain that arise when nodes update either the database or the curent variable instantiation. These dependencies are maintained if the nodes in a chain are executed sequentially, and the current variable instantiation is passed into and updated by each node of the chain as it executes.

Since we are only interested in finding one solution pyramid, the concurrent search strategy for pyramids described in this section is optimistic but precautious. Whenever a successful child is found, its parent optimistically assumes that the child is its one and only child in the solution pyramid. Being somewhat precautious, the parent suspends, rather than terminates, all concurrent searches for the siblings of the chosen child. In addition, the successful child suspends itself. If the search later discovers that the chosen child does not lead to a solution pyramid, the parent can resume the suspended searches. Furthermore, since there may be more than one instance of a node that leads to a solution, the chosen child's search is also resumed. Because searches are suspended and resumed, our search is not entirely concurrent and must employ some backtracking.

When a concurrent search of a pyramid encounters a parent-child hierarchy, it spawns a process to independently search each of the children. There are three types of search processes employed: two corresponding to the two types of concurrency, and one for the root PAN node. *Chain processes* sequentially execute the nodes in a chain for a given TIN. They are spawned by RAN processes and initiate all backtracking for the concurrent search. *RAN processes* execute RANs and are spawned either by the root PAN process or a chain process.

Before describing PCS, it is helpful to analyze the possible execution states of a process. A process is in one of three states as depicted in Figure 4, which shows the states for a

process P with parent $parent(P)$. When a process is spawned, it is initially *active*. An active process searchs for a partial-solution to its assigned portion of the search structure. If it finds one, it returns this solution to its parent and suspends itself - i.e. its state becomes *suspended*. An active process also becomes suspended if some other child is chosen as its parent's solution child. A suspended process is known to the system and can become active again at any time; it is reactivated by its parent. Before a process is suspended, it must return a *resume-state* to its parent that contains the information that is needed to resume its search. A process is *dead* when it no longer has the potential to find more solutions. This happens when it has exhausted all possible searches or when it is terminated by its parent.

Having introduced these states, we describe the relationship between processes and their states. Every process except the root process has a parent process. The parent of a chain process is a RAN process, and the parent of a RAN process is either a chain process or the root process. The relationship between the possible states of the nodes in a hierarchy exhibit the following properties:

- all descendents of a suspended node are either suspended or terminated,
- all descendents of a terminated node are terminated,
- a parent of an active node is always active, and
- descendents of an active node can be either active, suspended, or terminated.

We now describe the basic execution of a PCS. It begins at the root node PAN of the pyramid. This root process spawns a RAN process for each of the PAN's RANs. The input for each spawned RAN process is the substitution of values for variables that represents the user-activated PAN's input parameters. If all of the RANs fail, then there is no solution pyramid and the user request is rejected. If one of the RANs succeeds, a solution pyramid has been found. The user request is satisfied, and the successful RAN's substitution is used to synthesize the return values of the root PAN.

Each RAN process initializes its substitution to the input values supplied by its parent process, which is either a chain process or the root process. Given this substitution, it then evaluates its condition against the database. It spawns a chain process for each qualifying tuple, passing in the substitution. If all the chains fail, then the RAN reports failure to its parent process and terminates.

A solution for the RAN has been found if one of the RAN's chains succeeds. The RAN immediately suspends all spawned chain processes that have not yet failed. It builds a RAN resume-state consisting of the set of chain resume-states, one for each suspended chain,

plus the resume-state for the chosen solution chain. It reports success to its parent process, returning as output both the RAN resume-state and the substitution of the successful chain. It then suspends itself so that it can be reactivated in the case that its parent chain process is forced to backtrack at some later time.

A chain process tries to find a complete chain for its TIN by sequentially executing the PANs and PINs of its chain. It maintains a substitution that is initialized by its TIN and the input substitution from its parent RAN. If it can successfully execute all the nodes in its chain, it has found a successful execution. It reports success to its parent RAN process, returning the final value of its substitution and its resume-state. This resume-state contains a *resume-stack* which is used during backtracking. It is empty when the process is initially spawned and maintained during the execution and reactivation of the chain.

Since concurrent search is not performed on pyramids that contain i/o-actions, the only type of PIN node that a chain process must execute are doit-actions. The chain process adds an undo-action for the PIN to the resume-stack that, when executed, undoes the effects of the PIN. Although doit-actions always succeed, they may be blocked by another transaction or another concurrent search within the same transaction. In this case, the chain process must wait. However, this is not considered a failure since the chain process will either eventually be allowed to proceed or, in the rare event that a deadlock, crash, or media failure occurs, it will be rolled back and restarted by the database concurrency control manager or the mechanism that provides mutual exclusion for the concurrent search.

A chain process executes a PAN node by spawning a RAN process for each of the PAN's RANs. If one RAN process returns successful, the chain process optimistically assumes that this RAN and its return substitution will lead to a successful completion of the chain. [5] The chain process suspends all of the PAN's spawned RAN processes that have not yet failed. It pushes a PAN resume-state on the stack. This resume-state consists of the set of RAN resume-states returned by the suspended RANs, a RAN resume-state for the successful RAN, and the current substitution. It then updates the current substitution according to the return values synthesized from the successful RAN's substitution.

Backtracking occurs when all RAN processes for the PAN return failure. When this happens, the process pops and performs all undo-actions that are on the top of the resume-stack until the stack is empty or until a PAN resume-state is found. This restores ("rolls") the database state back to the backtrack point. If the backtrack stack is empty, then the chain process has failed and unsuccessfully returns to its parent RAN process. Otherwise, it pops the PAN resume-state off the stack, sets the current substitution to the substitution

[5] At this point we could have chosen concurrently investigate all possible correct completions of the chain for each RAN process that returns successfully.

contained in the resume-state, and reactivates all the RAN processes whose RAN resume-state is in the PAN's resume-state.

At any point, a parent process can suspend any of its active subprocesses. When a RAN process is suspended by its parent chain process, it, in turn, suspends all its spawned chain processes that have not yet failed. It builds a RAN resume-state consisting of the set of chain resume-states returned by each suspended chain. When a chain's parent RAN process asks it to suspend itself, it suspends all currently spawned RAN subprocesses that have not yet failed. It pushes on the stack a PAN resume-state that contains the current substitution and a set of the RAN resume-states returned by the suspended RANs. It returns to the parent RAN a chain resume-state that consists of the resume-stack.

PCS depends on the ability to resume searches when RAN and chain processes are reactivated. A RAN process resumes its search by reactivating all suspended chain processes in the RAN resume-state. A chain process resumes its search by initially backtracking. Its resume-state contains a resume-stack whose top element is a PAN resume-state that contains the resume-states for all RANs that were currently executing when the chain process was suspended. This element is popped from the resume-stack, the chain's current substitution is re-initialized to the substitution contained in this PAN resume-state, and all the RANs that have a RAN resume-state in the PAN resume-state are reactivated.

When a solution for the user request is found, there must be a clean-up procedure that terminates all the suspended processes. There is no such procedure required when the user request fails since a failure implies that all sub-processes have failed.

It is possible that a more realistic execution strategy would exploit only rule concurrency and not tuple concurrency, since the expensive computation is the evaluation of conditions against the database. In such a strategy, a separate RAN processes would be spawned for each RAN of a PAN, but the RAN process would sequentially process each TIN and the TIN's corresponding chain. In such a strategy, the functionality of chain processes would be absorbed by the RAN processes.

4.2.1 Mutual Exclusion

The concurrent search strategy described assumes that the database updates of concurrently executing processes are mutually exclusive. In this subsection we briefly suggest two ways in which such exclusion can be obtained.

Nested Transactions

The first solution treats the concurrent processes as nested transactions [RM89, Mos81, Mos87, HR87] that compete with each other for access to the database. However, the

locking rules for nested transactions do not apply directly. The rules for releasing and inheriting locks differ due to the behavioral differences between a failed process and an aborted process. A *failed process* is one that aborts itself because it detects something in the database state or the user input that indicates that it should not proceed.[6] An *aborted process* is one that is aborted by the system because of a deadlock or fails due to a system crash or a media failure.

The modified locking strategy is as follows. Read-locks are obtained when a RAN process evalutes a condition, and write-locks are obtained when a chain process executes a doit-action PIN. Locks can only be obtained if the only other processes that hold the lock are ancestors of the requestor. Otherwise, the requestor must wait. When a subprocess is chosen as the successful child of its parent, the parent inherits all the locks of the subprocess. The subprocess releases all its locks, but keeps a record of them to use if it is reactivated during backtracking. All other suspended processes hold their locks until they succeed and are chosen or until they are terminated.

When a subprocess fails, its read-locks must be inherited by its parent process. They can only be released when a solution has been found for the root process. If no such solution exists, they must be held until the user transaction commits. However, the write-locks need not be inherited. These issues are further discussed in section 5. The failed subprocess releases all its locks and terminates. When bactracking occurs, suspended processes are reactivated as described before with the exception that the previously chosen child first reclaims the locks it previously held from its parent.

The advantages to this mechanism are that it can use the locking mechanisms provided by the underlying database system and it does not require any copying of page-tables or data items. However, this solution could potentially cause severe blocking, reducing concurrency to a point where the search is effectively being done sequentially. Furthermore, it is possible for a suspended process to hold a lock that is needed by an active process, blocking the active process indefinitely. If the active process is the pyramid's only opportunity for success when the suspended process is not chosen, a deadlock occurs. The suspended process will not release a lock until the search fails and backtracks. But the search will not fail and backtrack unless the active process obtains the desired lock. Therfore, in order to make this solution to mutual exclusion useable, no nodes can be suspended. When a solution for a RAN or PAN node is chosen, all competing processes must be terminated rather than suspended. If the execution backtracks to the RAN or PAN at some future point in the execution, all of the terminated processes must be restarted; all of the previous work

[6]This is referred to as program-enforced abort in [HR87].

performed by these processes is lost.

Shadow Paging

The second solution employs techniques similar to shadow-paging. Each spawned proccess records its updates locally. These updates are seen by all of the process' subproccesses, but are propagated to its parent only if the process is chosen as the parent's successful child.

The advantages of this solution are that it maximizes concurrency and does not redo any work. When necessary, previous computations of processes are reused. The problem is that it requires exponential overhead in copying and propogating database states between processes.

Obviously, the entire database need not be copied each time a process is spawned. Furthermore, chain process are the only processes that update the database, so updates need only be maintained and propagated by chain processes.

Each chain process maintains a local *delta-page* table that reflects any changes that have been made to the database since the root PAN was activated. If the chain's grandparent is also a chain, then its delta-page table is initially the value of its grandparent's page table. Otherwise, the grandparent is the root and the delta-page table is initially empty.

When a chain process chooses a successful RAN for the execution of a PAN, it replaces its delta-page table with the delta-page table of the chosen RAN's chosen chain. When a chain process executes a doit-action PIN, it must make this update visible to itself and all processes it subsequently spawns, but invisible to all concurrently executing searches. If the update is to be made to a logical page $LP1$ that is in the chain's delta-page table, then the update can be made directly to the page referenced by $LP1$. Otherwise, the physical page $P1$ that is pointed to by $LP1$ is copied into $P1'$. $LP1$ is updated to reference $P1'$ and added to the chain's delta-page table. The update can then be performed on the local copy of $LP1$.

In the concurrent search strategy described previously, a doit-action PIN must be undone when a chain backtracks over the PIN. However, the delta-page tables never need to be updated during backtracking; nor do they need to be saved with each PAN resume-state in the chain's resume-stack. When backtracking, a chain can only be successful if some PAN is found that has an alternative solution RAN. When this happens, the chain's delta-page table is replaced with that of the new solution RAN's chosen chain.

When a successful execution is found for the root, then the updates in the delta-page table for the chain of the successful RAN are applied to the database. All copies of pages for terminated processes should be garbage collected.

5 Concurrency Control Issues

The Update Dependency Language is executed in a subsystem of a traditional DBMS and any activation of a procedure is part of some user transaction, referred to as the *containing transaction.* Hence, a procedure activation executes concurrently with other database operations and other procedure activations. The correctness criteria for the concurrent execution of transactions, called *serializability*, requires that the effects on the database of any concurrent execution of a set of transactions can be obtained by some serial execution of the same set of transactions.

For the most part, concurrency control can be correctly handled by applying the existing mechanisms provided by the DBMS. We assume support for basic two-phase locking [EGLT76], in which locks are obtained on data items that are read or written. Concurrency control for a sequential search strategy, such as PDFS described in section 4.1 is the most direct application of the DBMS locking mechanism. Read-locks are obtained for all tuples accessed when computing the conditions of RANs and write-locks are obtained for all tuples updated by doit-action PINs.

The concurrency control mechanisms for the concurrent search strategy described in 4.2 depend on the type of mutual exclusion mechanism employed. If the subprocesses are run as nested transactions, then the locks are obtained as described previously. If the shadow-paging technique is used, then locks can be obtained using the DBMS lock manager. However, so that the subprocesses do not conflict with each other, all locks are obtained on behalf of the containing transaction. Read-locks are obtained from the DBMS as the logical pages are read. However, write-locks are only obtained when a solution to the root process is found and the updates are actually made to the DBMS. They are not obtained when the updates are made to the subprocess' local pages. Hence, only updates that persist obtain write-locks, and all write-locks are obtained together.

It may seem that this delay in obtaining write-locks may not guarantee serializability. But serializability can only be affected if a data-item that is read by the search is subsequently written by another transaction before the search completes. This will not happen since the read-locks are obtained as the conditions are computed. If another transaction is reading or writing a data-item that is to be written by the search, then the search must wait until the transaction releases the lock on the item.

Early release of locks

As described in section 4, not all nodes in a pyramid participate in solution pyramids; furthermore, only the effects of one solution pyramid persist after the procedure activation

completes. Therefore, there may be several data-items that are read or updated at nodes that do not participate in the selected solution pyramid. Consequently, there may be some data-items that are read- or write- locked by the containing transaction only at these nodes. We shall refer to such locks as *false* locks.

It may be possible to improve concurrency by releasing false locks before the containing transaction commits, as suggested for partial rollbacks [MHL+91]. In some cases, this can be done without jeopardizing serializabily. In the remainder of this section we show that it is always possible to release false locks early when a solution pyramid is found. However, unlike partial rollbacks, read-locks must be maintained either until a solution pyramid is found or, if there is no solution pyramid, until the containing transaction commits or aborts.

It is worth noting that a lock that is requested at a node that is not in the solution pyramid is not considered a false lock if it was obtained prior to the activation or from some other node that participates in the solution pyramid. Also, sometimes write-locks are obtained by escalating existing read-locks. If an escalated write-lock for data-item is only used by nodes that are not in the solution pyramid, we still refer to this lock as a false lock. However, when such a lock is released, we mean that it is demoted back to a read-lock.

When a search for a solution pyramid detects a failure, it can release (or demote to read) known false write-locks immediately, without jeopardizing serializability. The false write-locks protect data items that are only updated by the containing transaction during an unsuccessful search. If the search is depth-first, a failure results in a partial rollback to the database state prior to the most recent choice point. During this partial rollback, all database updates that resulted from the doit-action PINs between the choice-point and the failure are undone. Similarly, if the search is concurrent using locking for mutual exclusion, then the updates are undone when the chain processes backtrack. [7] Hence, all updates that occurred since the most recent branching node will never be performed on the database since they will not be part of the solution pyramid. In both cases, the update will not be seen as part of the containing transaction and will therefore not need to be serialized with other concurrent operations.

However, false read-locks cannot be released as soon as a failure is detected. All read-locks must be kept during the entire search for a solution pyramid, since this search makes decisions based on the data-items it reads (or does not read). If these false read-locks are released before a solution pyramid is chosen, other transactions can update data-items that would have otherwise been locked. These updates could potentially cause some of

[7] As described above, for concurrent searches that employ shadow-paging techniques, locks are never obtained for these updates since they are not part of the solution pyramid updates that are performed on the database.

the rejected sub-pyramids to be successful. However, the search will not reconsider the sub-pyramids and may incorrectly reject a valid update.

The problem is even worse if the other transactions also make updates that make it impossible for the search to find a solution pyramid. Now, this scenario is always possible since nodes can never be locked until they are searched. However, a correct transaction will likely only invalidate a solution pyramid if it makes updates that create new solution pyramids. So, it is possible for a concurrent execution of a set of transactions to give a value for the success or failure of a procedure activation which could never be obtained with a serial execution of the same set of transactions.

Since the update dependency procedure and the user transaction may make database updates based on the result of an procedure activation, this type of behavior is not desirable. It may lead to concurrent executions that are not serializable. Consider the following example.

Example 7 Suppose a database schema has one relation for each state that records information about the suppliers in that state. This schema may contain relations such as: `VaSupp(S#, ...)`, `MdSupp(S#, ...)`, ..., `CaSupp(S#, ...)`. Suppose that there is a view `LocShpmnt(S#, P#, Qty, Loc)` which records shipments whose supplier, `S#`, is local to the shipment's location `Loc`. Then the update procedure may contain the following two rules that apply to shipments that are located in Washington, DC. (For simplicity, assume that all the actions of each of these rules succeed.)

```
insert LocShpmnt(S# = S, P# = P, Qty = Q, Loc = L)
-> L = ''Washington, D.C.'' and VaSupp(S#), ... .                (r1)

-> L = ''Washington, D.C.'' and MdSupp(S#), ... .                (r2)
```

Consider the following database state and set of transactions.

```
DBstate:  supplier S1 is currently in MdSupp.

T1: tries to insert supplier S1 as a local supplier for a shipment
    which is located in Washington

T2: modifies the database to reflect the fact that S1 has
    moved from Maryland to Virginia with the following two actions:
```

```
(1) deletes S1 from MdSupp
(2) inserts S1 into VaSupp
```

Any serial execution of these two transactions will result in the granting of T1's request. If T1 is executed before T2, then the request will be granted by a successful application of rule r2. If T2 is executed before T1, then the request will be granted by a successful application of rule r1. However, the following sequence of events shows a concurrent execution that is permissible if read-locks along failed paths are released.

```
(1) T1:  tries to insert supplier S1 as a local supplier for a shipment
         which is located in Washington

(2) T1:  activates procedure ''insert LocShpmnt''

(3) T1:  tries r1; r1 fails since S1 is not in VaSupp

(4) T2:  delete S1 from MdSupp

(5) T2:  insert S1 from VaSupp
         /* T2 could not do this if the read-lock on VaSupp was kept */

(6) T2:  commits and releases all locks

(7) T1:  tries r2; r2 fails since S1 is not in MdSupp
```

This execution results in the rejection of T1's request; it is not equivalent to either of the serial executions.

■

Since false read-locks keep other transactions from updating the database in such a way that makes the decision to reject a path incorrect, they should be released only when a solution pyramid is found for the user-activated update procedure. If there is no solution pyramid, all read-locks must be kept until the containing transaction commits or aborts.

Suppose transaction T replaces one correct solution pyramid with another (like transaction $T2$ above). Suppose transaction S is performing a search for a correct solution

pyramid. If S keeps its false read-locks, the locks prevent T from making solution pyramids that include the locked nodes. If T creates new solution pyramids before invalidating old ones, then T is blocked until S completes. If T invalidates the old solution pyramids first, then, in the worst case, a conflict between T and S will result in a deadlock, which will be resolved by the underlying lock manager. In the above example, `T2` would wait at step `5` until `T1` completes.

When a solution pyramid is found for a user request (i.e. a root node PAN), all false-locks, both read and write, can be released. Recall from section 4 that, to the user transaction, the sequence of actions and queries that a successful update procedure activation performs corresponds to a left-to-right traversal of the leaf nodes of the chosen solution pyramid. TINs represent database reads, and PINs represent either database updates or i/o-actions. Therefore, locks obtained on the solution pyramid must behave as if they were obtained directly by the transaction and not through the activation. However, the false-locks can be safely released.

In summary, for procedure activations that occur in a DBMS that supports two-phase locking, false write-locks can be released as soon as they are detected, and false read-lock can be released only when the request is granted.

6 Related Work

The oldest and best known technique for supporting constraints in database systems is database normalization, which results in database designs that enforce some functional dependencies through the use of keys in the model [Cod72, Cod74, Arm74, Ber76, Fag77, Fag79].

However, several applications require constraints that are too complex to be represented by relations and keys alone. Hence, a number of extensions to the relational model and semantic data models have been defined. Most of these extensions and new models provide different types of relations with different update semantics [SS76, Che76, BFM79, Cod79]. Some of these extensions and new models also provide a variety of types of constraints on domains (e.g. no-nulls, subrange) and between active domains (e.g. equality, subset, exclusion, partition).

Several proposals include a general constraint definition capability; some extend SQL [EC75] and some use general predicate logic [HM76] or first-order logic [GJ82] to formulate constraints. A general problem with most of these proposals is that they are prohibitively inefficient to use during database update, and substantial efforts have been invested in trying to make them more efficient [Sto75, HS78, BP79].

The notion of a database transaction [Gra78] was introduced to allow the grouping of several database updates, suspending integrity constraint checking until the transaction commits. Hypothetically, all the integrity rules would be explicitly represented in the database and the database system would check that the database state is consistent with the integrity constraints at the end of each transaction. Since only few database systems support any integrity constraints, transactions are usually used to group a set of updates which must be executed together. In this environment, the integrity of the database is fragile since the logic for enforcing constraints is replicated in each application. A system based on update dependencies replaces commit-time integrity constraint checking.

Update dependencies are closely related to a class of concepts which includes triggers [Esw76], alerters [BC76], condition-actions [HM76], and event procedures [BFM79]. These concepts originated from early work on integrity constraints and have had a substantial impact on the database area. The basic idea behind this work is that the database monitors database activity and executes a corresponding action when certain conditions on the database state are met. If the condition that triggers the action is a violation of an integrity constraint, then the database administrator can program the action to either reject the offending update request or to perform some corrective-actions that make the new database state consistent.

Our work has been influenced by related work on transaction specification (as found in the TAXIS system [Bro81,MBW80]), goal oriented approaches to constraint statisfaction [SK84], and early work on providing a Prolog front-end to relational systems [JCV84]. Our language is largely rule-based, but, unlike other rule-based languages that have been integrated into databases [DE89,dMS88,Han89,MD89,SPAM91,SLR88,SJGP90,WF90], it is goal-oriented. Its purpose is to support update dependencies, not to provide a general rule-system environment. We feel, for integrity constraint maintenance, a goal-oriented approach is preferable because it allows and requires the user to integrate all the update dependencies for an update into the same procedure. However, the integration is modular in the sense that only the update dependencies that have the same condition must be integrated together in the same rule.

The concept of defining higher abstractions on existing models such as the relational systems is in line with that of the direction taken in [SAHR84,TZ84,Zan83]. The use of abstractions to maintain semantic integrity constraints has been influenced by the constraint connections of the Structural Model [WE80].

Our formalism is related to various update semantics that have been defined in logic programming. We extend the algebraic semantics of updates presented in [AG85], i.e. updates are viewed as mapping between database states. As in [Nic82], we assume that the

database is in a valid state, satisfying all integrity constraints, at the beginning of each update procedure activation. Our update procedures are similar to the update procedures of DLP, [MW87,Man89] and the update predicates of LDL [NT89]. All of these formalisms assume an immediate update policy (i.e. a transaction can see its own updates) and support hypothetical reasoning. In LDL, there is no execution order among update predicates for rules that satsify the Church-Rosser property. However, they do not have provisions for undoing updates during backtracking. They also only allow update operations for base relations. DLP, on the other hand, allows update operations for views and undoes updates during backtracking. Like Update Dependency procedures, there is an implied order between updates in the rules. Both DLP and LDL serve as a uniform interface to a logic database, and updates to base relations can be issued from any query. The focus of their work is to provide semantics for updates in pure prolog and LDL respectively. On the other hand, our language focuses more on the interaction between user requests and the actual operations that are performed on the physical database in an effort to providing a means for specifying the correct evolution of the database. We therefore have update procedures for all base table updates in addition to views. Updates to the database are only allowed through the update's corresponding procedure.

In addition to supporting integrity constraint maintenance, our language also inherently supports (because it is based on update dependencies) the specification of view update policies and view maintenance. In systems that do not support updates to views, our language can be used to specify view update algorithms for theoretically updatable views[FC85,BS81]. It can also be used to specify updates for non-theoretically updatable views by incorporating domain-specific knowledge as suggested in [FC85,Kel85].

7 Conclusions and Future Work

In this paper, we formally defined a language which supports update dependencies in the relational model. We demonstrated its use with numerous typical database examples. We described two different strategies, depth-first and concurrent search, for executing procedures of this language, and we developed new modified two-phase locking strategies with early write lock release for executing these procedures in a traditional database environment.

The Update Dependency Language has been used to specify interoperability in engineering information systems [RMSF91] and multidatabases [LMR90]. We have also implemented a prototype interpreter for the Update Dependency Language in Prolog, which is currently being used in a joint project with The Mechanical Engineering Department at Maryland [HM89]. The purpose of this project is to develop, validate and test oper-

ational specifications of a Computer Integrated Manufacturing system that integrates the CAD/CAPP/MRP II Systems. The mechanical engineers have written a large set of modification procedures, and have been able to specify all of the selected operations in their CIM system with only small amount of consultation from us.

We are currently approaching the implementation of this language from two angles. First, we are building an interpreter, UDappl, as an application which issues SQL queries and modifications to a commercial DBMS (we are currently using Oracle). The interface to UDappl will allow the user to issue modification requests in a form similar to the req-actions described in section 2.3. This is the next generation of interpreter that will be used in the CIM project, and is useful because it is easily ported to any commercially available DBMS with an SQL query language.

Secondly, as described in this paper, we plan to integrate a subsystem for executing modification procedures into a DBMS, such as ADMS [NR91]. This will allow the modification procedures to truly guard their relations and to be activated by set-oriented modifications which are typically issued from SQL. We will also be able to experimentally investigate the concurrency issues for the different control strategies described in sections 4 and 5. Furthermore, the implementation must consider efficiency and pragmatic issues involved in implementing a system that backtracks and undoes modifications and processes one tuple at a time in search for a solution. We are currently exploring the use of incremental access methods in backtracking.

In addition to building interpreters for this language in a centralized environment, we would like to investigate how the update dependency concept can be adapted to decentralized client-server architectures [DR91]. For example, in a workstation-server architecture, it is too restrictive to have the server delay operation while waiting for a workstation. However, it seems reasonable to have a workstation wait for the server. These types of observations will require a slightly modified semantics for the modification procedures.

Acknowledgements

We would like to thank Jennifer Widom, Jorge Lobo, Raymond Ng, Louqia Raschid and Lynn Apseloff for giving us feedback on this work.

References

[ACL91] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule system. In VLDB [VLD91], pages 479–487.

[AG85] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *Proc. of the 11th Int. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985.

[Arm74] W. W. Armstrong. Dependency structures of data base relationships. In IFIP [IFI74], pages 580–583.

[BC76] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational data bases. Technical Report 76-10-02, Department of Decision Sciences, Wharton School, University of Philadelphia, Philadelphia, Pennsylvania, November 1976.

[Ber76] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Trans. on Database Systems*, 1(4):277–298, 1976.

[BFM79] B. Breutman, E. Falkenberg, and R. Mauer. CSL, a language for defining conceptual schemas. In G. Gracchi and G.M. Nijssen, editors, *Database Architecture*. North Holland Publishing Company, 1979.

[BL88] H. Boral and Per-Ake Larson, editors. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, June 1988.

[BP79] Badal and Popec. Cost and performance analysis of semantic integrity validation methods. In SIGMOD [SIG79], pages 9–36.

[Bro81] M. L. Brodie. On modeling behavioral semantics of data. In *Proc. 7th Int. Entity-Relationship Conf.*, Cannes, France, September 1981.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. on Database Systems*, 6(4):557–575, December 1981.

[Che76] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[CLM89] J. Clifford, B. Lindsay, and D. Maier, editors. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, June 1989.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, Courant Computer Science Symposium, Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N. J., 6th edition, 1972.

[Cod74] E. F. Codd. Recent investigations into relational data base systems. In IFIP [IFI74].

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):397–434, December 1979.

[Coh89] D. Cohen. Compiling complex database transistion triggers. In Clifford et al. [CLM89], pages 225–234.

[CW89] S. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proc. 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* [POD89], pages 200–209.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In VLDB [VLD91], pages 577–589.

[Dat86] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[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 – Proc. from the Second Int. Conf.*, pages 333–351, Redwood City, California, 1989. Benjamin/Cummings.

[dMS88] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proc. of the 4th Int. Conf. on Data Engineering*, pages 234–241, Los Angeles, California, February 1988.

[Don89] G. Dong. On distributed processibility of datalog queries by decomposing databases. In Clifford et al. [CLM89], pages 26–35.

[DR91] A. Delis and N. Roussopoulos. Performance comparison of three modern dbms architectures. Technical Report CS-TR-2679, University of Maryland, College Park, Maryland 20740, May 1991.

[EC75] K. Eswaran and D. Chamberlin. Functional specifications for a subsystem for data base integrity. In *Proc. of the 1st Int. Conf. on Very Large Data Bases*, Framingham, Massachusetts, September 1975.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Comm. ACM*, 19(11):624–633, November 1976.

[Esw76] K. Eswaran. Aspects of a trigger subsystem in an integrated data base system. In *Proc. 2nd Int. Conf. on Software Engineering* [SE776].

[Fag77] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. on Database Systems*, 2(3):262–278, 1977.

[Fag79] R. Fagin. Normal forms and relational database operators. In SIGMOD [SIG79], pages 9–36.

[FC85] A. L. Furtado and M. A. Casanova. Updating relational views. In Won Kim and David S. Reiner adn Don Batory, editors, *Query Processing in Database Systems*. Springer-Verlag, Berlin, 1985.

[GJ82] J. Grant and B. E. Jacobs. On the family of generalized dependency constraints. *Journal of the ACM*, 29:986–997, 1982.

[GMJ90] H. Garcia-Molina and H. V. Jagadish, editors. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, June 1990.

[Gra78] J. Gray. Notes on database operating systems. Technical Report RJ 2188 (30001), IBM Research Laboratory, San Jose, California, February 1978.

[GST90] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of datalog queries. In Garcia-Molina and Jagadish [GMJ90], pages 143–152.

[Han89] E.N. Hanson. An initial report on the design of ariel: A DBMS with an integrated production rule system. *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems*, 18(3):12–19, September 1989.

[HM76] M. Hammer and D. McLeod. A framework for database semantic integrity. In *Proc. 2nd Int. Conf. on Software Engineering* [SE776].

[HM89] G. Harhalakis and L. Mark. A knowledge-based prototype of a factory-level cim system. *Journal of Computer Integrated Manufacturing Systems*, 2(1):11–20, 1989.

[HR87] T. Härder and K. Rothermel. Concepts for transaction recovery in nested transactions. In U. Dayal and I. Traiger, editors, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 239–248, San Francisco, May 1987.

[HS78] M. Hammer and Sarin. Efficient monitoring of database assertions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Austin, Texas, May 1978.

[IFI74] *Proc. 1974 IFIP Congress*, Amsterdam, 1974. North Holland.

[JCV84] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing prolog front-end to a relational query systems. In Yormark [Yor84], pages 296–306.

[Kel85] A. Keller. Algorithms for translating view updates to views involving selections, projections, and joins. In *Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[KKM83] S. Kasif, M. Kohli, and J. Minker. PRISM: A parallel inference system for problem solving. In *Logic Programming Workshop*, pages 123–152, Praia da Falesia, Algarve, Portugal, 1983.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[LS86] E. Shapiro L. Sterling. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[Man89] S. Manchanda. Declarative expression of deductive database updates. In *Proc. 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* [POD89], pages 93–100.

[Mar85] L. Mark. *Self-Describing Database Systems - Formalization and Realization.* Ph.D. dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1985. TR-1484.

[MBW80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A language facility for designing interactive database-intensive applications. *ACM Trans. on Database Systems*, 5(2):185–207, June 1980.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In Clifford et al. [CLM89], pages 215–224.

[MHL+91] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 1991.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Boston, MA, April 1981. MIT/LCS/TR-260.

[Mos87] J. E. B. Moss. Log-based recovery for nested transactions. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 427–432, Brighton, England, September 1987.

[MW87] S. Manchanda and D. S. Warren. A logic-based language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 10, pages 363–394. Morgan-Kaufmann, Los Altos, California, 1987.

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

[Nil80] N. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, California, 1980.

[NR91] A. Stamenas N. Roussopoulos, N. Economou. Adms: A testbed for incremental access methods. *IEEE Trans. on Knowledge and Data Engineering*, 1991.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[POD89] *Proc. 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Philadelphia, PA, 1989.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proc. of the 15th Int. Conf. on Very Large Data Bases*, Amsterdam, August 1989.