The Immutable Attributes Myth

Applying Transition Constraints, Referential Actions and Permissions without Them

Adrian Hudnott, Jane Sinclair, and Hugh Darwen

University of Warwick, Coventry, CV4 7AL, UK

Abstract. We consider the difference between insertion with simultaneous deletion versus an UPDATE that makes identical changes. A transition constraint must not be satisfied for one of these and not the other, because if it were then it could be circumvented by users restating UPDATES as inserts and deletes or vice versa. The same difficulty arises with referential actions, triggered procedures and security permissions. We introduce a method that works independently of the language keywords used to find pairs of tuples that are interpretable as one being updated to become the other. We propose that enterprises decide when a change that is interpretable as an UPDATE should be interpreted as one, and when it should be interpreted as DELETE and INSERT. We propose enterprises codify their policies as part of database construction.

Keywords. deferred constraint checking, dynamic integrity constraints, lineage, multiple assignment, transition constraints

1 Introduction

Transition constraints restrict the permitted updates based on the current and historical contents of the database [12]. They are additional to ordinary database integrity constraints. Transition constraints refer to the database contents at two or more points in time, unlike ordinary integrity constraints. Our running example is that a person cannot go from being single to being widowed without having been married in between. Database updates are destructive, so constraints involving states additional to the updated state and the one immediately prior to update require extra storage to preserve the extra states and a more complex database language to refer to them. That task is informationally equivalent to a temporal database where the transition constraints in the non-temporal database design map onto ordinary integrity constraints in the temporal one, as noted by Gertz and Lipeck [8]. Darwen, Date and Lorentzos [6] provide a temporal database model that permits declaration of such constraints. Therefore, an area of particular interest is transition constraints that refer to the updated state and the immediately previous one only, because these do not require temporal DBMS functionality. Many applications do not need the extra flexibility, complexity and storage overhead that accompanies a temporal database but nonetheless may have transition constraints.

We show that unique interpretation of a transition between database states is difficult and propose a way to solve the problem. In addition to transition constraints, application of referential actions, triggered procedures and permissions also depends on interpretation of transitions. SQL does not support transition constraints declaratively but the issues remain applicable if they are implemented using triggered procedures, as suggested by Koppelaars and De Haan [9].

2 Ambiguous Updates

A database relvar[1] is updated using the INSERT, DELETE, UPDATE or assignment (`:=`) updating operators. Let "update" refer to invoking one or more updating operators and UPDATE refer to the updating operator with that name. Despite informal talk that a particular attribute `A` is, "Not subject to update", no attribute is ever updated. The meaning is that the UPDATE operator is never invoked in a way where the projection of the relvar over `A` differs before and after the update. Informally saying a tuple is "updated" means the tuple satisfies a specified WHERE clause and the relvar is updated to remove the tuple and add another tuple that is sufficiently similar to the removed tuple to be perceived as the removed tuple being edited to become the added tuple. Each tuple in a relation assigned to a relvar corresponds to a proposition about an *entity* (such as an employee) or a *relationship* between entities, as conveyed by familiar ER-models [3]. Tuples are "sufficiently similar" if they denote propositions about the same entity or relationship.

Example 1 (UPDATE vs. INSERT-DELETE). Suppose a database relvar named `EMP` has the predicate, "*Name* of *Address* is an employee with marital status *Status*", and that `EMP` initially has the value shown in Fig. 1. The TID column is not part of the relvar. It denotes the location where each tuple is stored and is visible only to the system itself. Assume there is a constraint in effect that no two employees have the same name and live at the same address. What is the user asserting if they issue, `UPDATE EMP WHERE Name = 'Jane Jones' AND Address = '16 Cherwell Close' ( Status := 'Widow' )`? Is it that the employee who was called Jane Jones and lived at 16 Cherwell Close is still an employee, is still called Jane Jones, still lives at 16 Cherwell Close and has changed her marital status from single to widowed? Or is it an assertion that the employee who was called Jane Jones and lived at 16 Cherwell Close is no longer an employee and another person, also called Jane Jones, has become an employee, also lives at 16 Cherwell Close and is a widow? The first interpretation fails a transition constraint that says that a single person cannot become a widow without first being married, but the second interpretation satisfies the constraint.

Is the intention to alter an existing employee record (the first interpretation) because the operator is UPDATE? If so, then the second interpretation would be implied by Listing 1.1.

[1] Short for "relation variable". Loosely a database relvar is a "database table"

TID	Name	Address	Status
1	John Smith	57 Green Lane	Single
2	Jane Jones	16 Cherwell Close	Single
3	Jane Jones	35 London Road	Widowed

Fig. 1. Initial value of the `EMP` relvar (table) plus tuple IDs (in margin)

Listing 1.1. INSERT and DELETE

```
INSERT EMP RELATION{ TUPLE{ Name 'Jane Jones', Address '16
   Cherwell Close', Status 'Widowed' } },
DELETE EMP WHERE Name = 'Jane Jones' AND Address = '16
   Cherwell Close' AND Status = 'Single';
```

Note the multiple assignment: two update operations executed as an atomic statement, constraint checking being performed at the statement boundary (indicated by the semicolon)[2] [5]. An SQL system would use deferred constraint checking for the same purpose, though not all SQL constraints are necessarily DEFERRABLE [11]. Therefore, non-deferrable constraints limit accidental exposure to the problem, though a determined user can delete more data and later re-insert it to make the change even with non-deferrable constraints. If the choice of updating operator disambiguates then consider Listing 1.2[3].

Listing 1.2. UPDATE expressed as INSERT and DELETE

```
INSERT EMP EXTEND (EMP WHERE Name = 'Jane Jones' AND Address
   = '16 Cherwell Close'){ALL BUT Status} ADD ('Widowed' AS
   Status),
DELETE EMP WHERE Name= 'Jane Jones' AND Address = '16
   Cherwell Close' AND Status = 'Single';
```

Does this assert "A new employee has been hired. She is just like the old employee, Jane Jones of 16 Cherwell Close (who has simultaneously ceased being employed), in every way except that she is a widow"? Or is it an UPDATE in disguise, by a user trying to bypass the transition constraint? Conversely, sometimes users write UPDATE as shorthand for deleting an entity and inserting a similar one. We conclude that updating operator names are insufficient to disambiguate.

Example 2 (Swapped Tuples). Consider:

```
UPDATE EMP WHERE Name = 'Jane Jones' AND Address = '16
   Cherwell Close' ( Address := '35 London Road' ),
UPDATE EMP WHERE Name = 'Jane Jones' AND Address = '35 London
    Road' AND Status = 'Widowed' ( Address := '16 Cherwell
   Close', Status := 'Single' ),
UPDATE EMP WHERE Name = 'Jane Jones' AND Address = '35 London
    Road' ( STATUS = 'Widowed' );
```

[2] In this example multiple assignment is not needed if the DELETE is done first but reordering is not always possible.

[3] Braces indicate projection.

EMP retains its existing value but nevertheless the Jane Jones originally of 16 Cherwell Close has gone from being single to being widowed, which we want to prevent. Therefore, while updating operator names are not a suitable basis for interpretation, comparing pre- and post-transition values alone is not either.

Example 3 (Assignment). A relvar can be assigned to like any other variable. INSERT, DELETE and UPDATE are shorthand for particular assignments [5]. For example, `UPDATE V WHERE p ( a1 := e1,... an := en )` can be rewritten as:

```
V := (V WHERE NOT(p)) UNION ((EXTEND (V WHERE p)
   ADD (e1 AS a1′,... en AS an′)){ALL BUT a1,... an}
   RENAME (a1′ AS a1,... an′ AS an))
```

Listing 1.3. Assignment

```
EMP := (EMP MINUS (EMP WHERE Address =  '16 Cherwell Close'))
     UNION (EXTEND (EMP WHERE Name = 'Jane Jones'){Name,
    Address} ADD ('Widowed' AS Status));
```

Consider Listing 1.3. How can transition constraints be checked for arbitrary assignments? How do we ensure that checking an INSERT, DELETE or UPDATE gives the same result as checking an equivalent assignment? Decomposing assignment into INSERTs, DELETEs and UPDATEs is important if cascaded operations or triggered procedures are specified, and for performance reasons too [10].

Example 4 (Non-SQL Security Permissions). Suppose Fred has permission to change employee details and insert new employee records but cannot delete records. The DBMS is instructed to reject DELETE invocations from Fred that target EMP but he can still remove an employee by executing, `UPDATE EMP WHERE Name = 'John Smith' ( Name := 'Jane Jones', Address := '16 Cherwell Close' )`, if UPDATE is defined as shorthand for an assignment as suggested by Example 3. A similar statement in SQL would raise a key constraint violation however, because SQL defines UPDATE differently.

Example 5 (SQL Security Permissions). If Fred has insert and delete permissions but not the UPDATE permission then he can nonetheless achieve the effect of an UPDATE using INSERT and DELETE, provided the intermediate state satisfies all non-deferrable constraints.

3 Surrogate Keys

EMP is unrealistic. Typically, the relvar would have a surrogate key, say EMP# (Fig. 2). According to Date, one feature of a surrogate is, "When a new entity is inserted into the database, it's assigned a surrogate key value that's never been used before and will never be used [assigned] again" [7]. `UPDATE EMP_SK WHERE Name = 'Jane Jones' AND Address = '16 Cherwell Close' ( Status := 'Widow')` cannot mean that the existing Jane Jones has been replaced by another Jane Jones because the EMP# value of the new tuple is 002 and a new Jane Jones would have a different EMP# value. However, the requirement that EMP# cannot be updated is currently unenforceable because updating EMP# is indistinguishable from firing an employee and simultaneously hiring another, unless the choice of updating operator disambiguates, which we argue against. Are surrogate keys a possible exception? No, because contrary to Date sometimes updating a surrogate attribute is required to protect anonymity that would otherwise be compromised by an unchangeable identifier, for example as part of witness protection. Surrogacy and immutability are distinct concepts. Furthermore, should a DBMS mandate surrogate keys for relvars in order to enforce transition constraints when those relvars would not need surrogate keys for any other purpose? We do not answer this moral question but instead provide an approach that works irrespective of the answer.

TID	EMP#	Name	Address	Status
1	001	John Smith	57 Green Lane	Single
2	002	Jane Jones	16 Cherwell Close	Single
3	003	Jane Jones	35 London Road	Widowed

Fig. 2. Adding employee numbers: EMP_SK

4 Source Extended Query Result

The proposed solution relies on knowing *why* a particular tuple appears in a query result, or in other words which tuples in the database contribute to each tuple appearing in the result? This is known as the query's lineage. We define an operator called the Source Extended Query Result (SEQR) to compute the lineage. The SEQR includes every attribute of its argument and an additional attribute called _TIDS[4] that holds the set of tuple IDs (TIDs) that comprise the lineage. DBMSs often use a tuple ID to locate a representation of the tuple on disk, such as Ingres for example [14]. SEQR uses tuple IDs to refer to tuples present before an assignment. Although verbose, the tuples could equivalently be written in place of their IDs.

The Source Extended Query Result of an expression E with respect to a relvar V is a lineage measure with three distinct features. Firstly, for every tuple $t \in$ E there are one *or more* tuples in SEQR(E, V), each corresponding to a unique tuple derivation. Some previous works also keep multiple derivations separate, such as Buneman et al.'s work using a tree data model [2], but many authors do not, such as Cui, Widom and Wiener [4] and Cheney et al. [1]. Secondly, only tuples contained in V are included in our measure, which is advantageous because other relvars are irrelevant in determining UPDATES to V. On the other hand,

[4] We adopt a convention that attribute names beginning with an underscore are reserved for system use.

restricting information to one relvar diminishes utility for answering more usual lineage related questions. Thirdly, SEQR tracks the lineage of a set difference more precisely than uniformly adding all tuples in the subtrahend to the lineage.

All views referred to in `E` are recursively replaced by their definitions prior to computing `SEQR(E, V)`.

Relvars. `SEQR(V, V)` is the extension of `V` with each tuple having a singleton set for its _TIDS attribute containing the TID of the tuple being extended. `SEQR(V1, V2)` for distinct relvar names `V1` and `V2` is `V1` extended with _TIDS being the empty set for every tuple.

Literals. `SEQR(E, V)` where `E` is a literal (for example, `RELATION{ TUPLE{ X 1 } })` is `E` extended with _TIDS being the empty set for every tuple.

Restriction. `SEQR(E WHERE C, V)` is `SEQR(E, V) WHERE C`.

Projection. `SEQR(E{A1,... An}, V)` is `SEQR(E, V){A1,... An, _TIDS}`. If `{A1, ... An}` is not a superkey of `E` then for any tuple in `E{A1,... An}` there may be multiple tuples in the SEQR, one for each way that the projected tuple can be *derived.*

Join. `SEQR(E1 JOIN E2, V)` is:

```
(EXTEND ((SEQR(E1, V) RENAME (_TIDS AS _TIDS1)) JOIN
   (SEQR(E2, V) RENAME (_TIDS AS _TIDS2)))
ADD (_TIDS1 UNION _TIDS2 AS _TIDS)){ALL BUT _TIDS1, _TIDS2}
```

When two tuples are joined every tuple that contributed to either of the tuples being joined contributes to the joined tuple.

Union. `SEQR(E1 UNION E2, V)` is `SEQR(E1, V) UNION SEQR(E2, V)`. If a tuple appears in `E1` and `E2` then it will have two corresponding tuples in the SEQR if and only if the tuple has different lineages in `E1` and `E2`.

Rename. `SEQR(E RENAME (A1 AS B1,... An AS Bn), V)` is:
`SEQR(E, V) RENAME (A1 AS B1,... An AS Bn)`.

Extension. In most cases `SEQR(EXTEND E ADD (exp AS A), V)` is `EXTEND SEQR(E, V) ADD (exp AS A)`. However, if `exp` is relation-typed then `SEQR(exp, V)` is calculated for each tuple in `E` and the lineage of the extended tuple is the union of the lineage of the unextended tuple and the lineage of every tuple belonging to the relation-typed attribute. Aggregation is shorthand for extension with a relation-typed attribute, so `SUMMARIZE EMP BY{STATUS} ADD (COUNT() AS Num)` is shorthand for:

```
(EXTEND (EXTEND EMP{Status} ADD (((EMP RENAME (Status AS
    Status2)) WHERE Status = Status2) AS Emps))
ADD (COUNT(Emps) AS Num)){Status, Num}
```

Figure 3 shows the SEQR of the inner EXTEND invocation with respect to `EMP`. `EMP` has two tuples with `Status` “Single”, hence `Emps` has two tuples in the extension of `TUPLE{Status 'Single'}` and those two tuples are formed by renaming and restricting `EMP`, hence contributing one TID each to the lineage of the extended tuple.

Status	Emps			_TIDS
Single	Name	Address	Status2	{ 1, 2 }
	John Smith	57 Green Lane	Single	
	Jane Jones	16 Cherwell Close	Single	
Widowed	Name	Address	Status2	{ 3 }
	Jane Jones	35 London Road	Widowed	

Fig. 3. `SEQR(EXTEND EMP{Status} ADD (((EMP RENAME (Status AS Status2)) WHERE Status = Status2) AS Emps), EMP)`

Difference, Antijoin. `SEQR(E1 MINUS E2, V)` depends on the structure of `E2`. Take the example `SEQR(E1 MINUS (E2 MINUS E3), V)` where `E1`, `E2` and `E3` do not include invocations of difference or antijoin. For an arbitrary tuple $t_1 \in$ `SEQR(E1, V)` corresponding to some tuple $t \in$ `E1` and having a _TIDS value L_1 there is a corresponding tuple in the result of `SEQR(E1 MINUS (E2 MINUS E3), V)` as follows. If $t \notin$ `E2` then t_1 appears in the SEQR.[5] If $t \in$ `E2` then for each $t_3 \in$ `SEQR(E3, V)` corresponding with t and having _TIDS value L_3, the tuple formed by extending t with a _TIDS value equal to $L_1 \cup L_3$ appears in the SEQR. In this case t appears in `E1 MINUS (E2 MINUS E3)` because it appears in `E1` and `E3`, preventing its elimination because of its appearance in `E2`. Thus all TIDs responsible for t's appearance in `E1` and `E3` are relevant, which is $L_1 \cup L_3$. The result generalizes: TIDS can contribute to the result if they contribute to an expression on the right-hand side of an even number of nested difference or antijoin invocations.

5 Identifying UPDATEs

The method begins by assuming an assignment does inserts and deletes only and then identifies insert-delete pairs that might alternatively be `UPDATE`d tuples. “Might be” because it is impossible to infer the user's intention from only the assignment and the existing database. The “actual” updates are those the user intended as updates, usually a subset of the “possible updates” identified by

[5] General lineage queries would demand recording the fact $t \notin$ `E2`. We have defined an extension to SEQR with two more underscored attributes for recording negative information (omitted because of lack of space).

the method. Sometimes it is impossible for the actual updates to include every possible update, for example:

```
UPDATE EMP_SK WHERE EMP# = '001' ( Status := 'Married' ),
INSERT EMP_SK EXTEND (EMP_SK WHERE EMP# = '001'){ALL BUT EMP
    #} ADD ('004' AS EMP#);
```

The updated value of EMP_SK has two tuples that derive from the existing tuple for employee 001. Did that tuple have the EMP# and Status attributes updated, or just Status? At most one tuple is the result of an UPDATE and the other is an INSERT but it is impossible to determine which one is which without relying on the updating operator name.

To apply transition constraints to the assignment V := E, using V to refer to the value of V before the assignment:

1. Identify tuples in V that are possibly UPDATEd: tuples in V but not in E *and* tuples in V and E but with different lineages. Let *sources* denote the set of tuples from SEQR(V, V) such that their projections over all but _TIDS are possibly UPDATEd tuples. If a tuple t appears in V and E then let t_v and t_e be corresponding tuples in SEQR(V, V) and SEQR(E, V) respectively. There may be multiple values of t_e if t is derivable in multiple ways. t_v is a source if and only there is no value of t_e such that _TIDS from t_e equals _TIDS from t_v, where _TIDS from t_v is the singleton containing the TID of tuple t in the physical representation of V. Loosely speaking, if t appears in the result of E and the DBMS determines this by knowing $t \in$ V and not accessing any other tuples in V then t has been "updated" by the identity function, which is not an UPDATE.
2. Identify the tuples in E that possibly result from UPDATES to V: tuples in E that have at least one corresponding tuple in SEQR(E, V) with a non-empty _TIDS value. Label the set of all such tuples from SEQR(E, V) the *outcomes*.
3. Identify some possible UPDATES: find pairs of tuples such that one is a source, one is an outcome, and they have the same _TIDS value.
4. For each source-outcome pair identified by step 3 remove the source from the set of sources and its TID from the _TIDS attribute of all outcomes.
5. Add the UPDATES found by step 3 to the set of all possible UPDATE pairs found so far.
6. Repeat steps 3–5 until no more pairs are found.
7. Compute the set of presumed inserted tuples and the set of presumed deleted tuples. Presumed inserts are tuples that are in E, not in V and not the result of a possible UPDATE. Presumed deletes are tuples that are in V, not in E and not the source of a possible UPDATE.
8. Check the transition constraints. Within a constraint declaration V refers to value of V before the assignment, V′ to the value after the assignment (which is denoted by E), V+ to the presumed inserts, V- to the presumed deletes, and V∼ to the possible UPDATES. V∼ has primed and unprimed attribute names for each attribute of V. Unprimed attributes hold values from the source tuple and primed attributes hold values from the outcome tuple.

If a transition constraint rejects a possible UPDATE then it reclassifies it as an insert and a delete (as will be shown with Example 4). Thus V+ does not necessarily include every "actual" insert and V- does not necessarily include every "actual" delete. Conversely, a constraint can use external knowledge to match a presumed delete and a presumed insert as an UPDATE that was not flagged as "possible", as will be demonstrated with an example with a surrogate key. If an outcome has multiple TIDS at step 7 (which can occur if an assignment includes a self-join of V) then it is ambiguous which tuple was UPDATEd to become the outcome projected over all but _TIDS. Step 7 classifies the outcome as a presumed insert. Further work could devise a language construct to permit the user defining the constraint to specify an alternative disambiguation strategy.

A transition constraint prohibiting a single employee from becoming widowed in one transition is expressed as:

```
TRANSITION CONSTRAINT NO_WIDOW_WITHOUT_MARRIAGE IS_EMPTY(
EMP∼ WHERE Status = 'Single' AND Status′ = 'Widowed' );
```

Applying the method to Listing 1.1 (INSERT and DELETE keywords), "E" is shown in Fig. 4.

Name	Address	Status
John Smith	57 Green Lane	Single
Jane Jones	16 Cherwell Close	Widowed
Jane Jones	35 London Road	Widowed

Fig. 4. Value assigned to EMP by Listing 1.1

sources = RELATION{ TUPLE{ Name 'Jane Jones', Address '16
    Cherwell Close', Status 'Single', _TIDS SET{2} } }
outcomes = RELATION{ TUPLE{ Name 'Jane Jones', Address '16
    Cherwell Close', Status 'Widowed', _TIDS SET{ } } }

The only tuple that could be the source of an UPDATE does not match the only one that could be the outcome of an UPDATE because $\{\} \neq \{2\}$, thus EMP∼ is empty. The DELETE and INSERT invocations are not regarded as UPDATE-ing the existing Jane Jones's record because the expression denoting what to insert does not refer to any part of the database. Values appearing in that expression and the database, such as "Jane Jones", are considered coincidental. Arguably the constraint can still be bypassed, though it is cumbersome to manually retype the value of every attribute. This problem will be addressed when surrogate keys are considered.

For Listing 1.2 (UPDATE expressed as INSERT and DELETE):

outcomes = RELATION{ TUPLE{ Name 'Jane Jones', Address '16
    Cherwell Close', Status 'Widowed', _TIDS SET{2} } }

sources is as before. The source tuple and the outcome tuple match as a possible UPDATE (Fig. 5). The transition constraint decides to interpret it as an UPDATE that is invalid. Listing 1.3 is equivalent to converting Listing 1.2 to assignment form and proceeds identically.

Name	Address	Status	Name′	Address′	Status′
Jane Jones	16 Cherwell Close	Single	Jane Jones	16 Cherwell Close	Widowed

Fig. 5. `EMP∼` for Listing 1.2

For Example 2 (swapped tuples) there are two sources and two outcomes and they match (Fig. 6). The constraint does not hold, as required.

Name	Address	Status	Name′	Address′	Status′
Jane Jones	16 Cherwell Close	Single	Jane Jones	35 London Road	Widowed
Jane Jones	35 London Road	Widowed	Jane Jones	16 Cherwell	Single

Fig. 6. `EMP∼` for Example 2

For Example 4 (non-SQL security):

```
sources = RELATION{ TUPLE{ Name 'John Smith', Address '57 Green
    Lane', Status 'Single', _TIDS SET{1} },
TUPLE{ Name 'Jane Jones', Address '16 Cherwell Close', Status
    'Single', _TIDS SET{2} } }
outcomes = RELATION{ TUPLE{ Name 'Jane Jones', Address '16
    Cherwell Close', Status 'Single', _TIDS SET{1} } }
```

John Smith matches with Jane Jones as a possible update to John's name and address. There is also a source for Jane Jone's record because the tuple is freed up by the UPDATE. However, it does not match any outcome. It is not presumed deleted because the same tuple is in `EMP` before and after the update.

For previous examples all of the possible UPDATES were accepted as actual UPDATES by the transition constraint. However, to enforce security permissions as they are commonly understood an interpretation must be imposed that prevents Fred from disguising deletions as UPDATES.

```
TRANSITION CONSTRAINT FRED_NO_DELETE WITH
   EMP- UNION (
      (EMP∼ MATCHING (EMP RENAME (SUFFIX '' AS '′')))
      NOT MATCHING
      (EMP∼{Name, Address, Status} RENAME (SUFFIX '' AS '′'))
   ){Name, Address, Status}
   AS DELETES : IS_EMPTY(DELETES) OR USER ≠ 'Fred';
```

That is, the expected interpretation of security permissions is that a tuple is "deleted" if (1) it is presumed deleted in the way we define the term, or (2) it is the source of a possible UPDATE and the result of the possible update was

present before the assignment and is not itself UPDATEd (or deleted but that is by-the-by).

Transition constraints can ensure the `EMP#` attribute of `EMP_SK` is immutable, if desired.

```
TRANSITION CONSTRAINT IMMUTABLE_EMP# IS_EMPTY(EMP_SK∼ WHERE
    EMP# ≠ EMP#′);
```

Other transition constraints involving `EMP_SK` need to include an expression amalgamating a deleted tuple and an inserted tuple with the same employee number with the possible UPDATES identified by the system.

```
TRANSITION CONSTRAINT NO_WIDOW_WITHOUT_MARRIAGE WITH
   EMP_SK∼ UNION (
      EMP_SK- JOIN (EXTEND (EMP_SK+ RENAME (SUFFIX '' AS '′'))
           ADD (EMP#′ AS EMP#))
   ) AS ACTUAL_UPDATES :
   IS_EMPTY( ACTUAL_UPDATES WHERE Status = 'Single' AND
      Status′ = 'Widowed' );
```

This constraint prevents any employee changing from single to widowed, even if the value of the `EMP#` attribute is manually typed into an INSERT instruction in a similar way to Listing 1.2.[6]

6 Conclusions and Further Work

Several common DBMS features depend on classifying changes into inserts, deletes and UPDATES. Surprisingly, we discovered the correct classification is not always obvious. The INSERT, DELETE and UPDATE keywords describe the changes needed to move between states but not the transition's meaning. Without support for multiple assignment or deferrable constraints every transition is achievable by exactly one updating operator invocation and can be assigned a unique interpretation. If multiple assignment or deferred constraint checking are used then there are multiple combinations of updating operators that produce the same transition and the transition does not have a unique interpretation.

We do not provide a way for the DBMS to infer meaning from update statements because the human brain is inaccessible. Our method analyzes the structure of the update statement and collates the pairs of distinct tuples where one is in the after state and was mechanically derived from the other, which is in the before state, hence the pair are a possible UPDATE. The intuitive notion of "mechanically derived" is formally defined by our SEQR function. An enterprise policy embedded in the transition constraints assigns unique meaning based on the possibilities inferred by the system. Codifying enterprise policy prevents circumvention of transition constraints that can otherwise occur if updating operator names are used to signify meaning.

[6] Unless all of the attributes are typed manually *and* the employee is *simultaneously* reassigned a *different* employee number. Then the INSERT would be interpreted as a request to insert a new employee.

Under our approach many constraints use WITH to declare a set of actual UPDATES different to the set of possibly ones identified by the system, that is they have an enterprise policy different to the default one. In practice, we expect the policy for a relvar will be consistent across different constraints, so some syntax separating the policy from the rest of the constraint is desirable. Shorthand syntax for common scenarios such as immutable attributes may also be helpful.

This paper was motivated by a larger transition constraints example [13] but the same problems apply to triggered procedures, referential actions and security permissions. An unknown number of existing applications have security holes or triggered procedures or referential actions that misfire or fail to fire because of the problem discussed. Therefore an investigation is advisable.

References

1. Acar, U.A., Ahmed, A., Cheney, J.: Provenance as dependency analysis. In: Arenas, M., Schwartzbach, M.I. (eds.) Proceedings of the 11th International Symposium on Database Programming Languages. pp. 138–152. Springer-Verlag, Germany (2007)
2. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and Where: A Characterization of Data Provenance, LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2001)
3. Chen, P.P.S.: The entity-relationship model — toward a unified view of data. ACM Transactions on Database Systems 1(1), 9–36 (1976)
4. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Transactions on Database Systems 25(2) (2000)
5. Darwen, H., Date, C.J.: Databases, Types, and the Relational Model: The Third Manifesto. Pearson Education, USA, 3rd edn. (2006)
6. Darwen, H., Date, C.J., Lorentzos, N.A.: Temporal Data and the Relational Model. Morgan Kaufmann, USA (2003)
7. Date, C.J.: Object Identifiers vs. Relational Keys, chap. 12, pp. 457–475. Relational Database Writings, Addison-Wesley Longman, Singapore (1998)
8. Gertz, M., Lipeck, U.W.: Deriving optimized integrity monitoring triggers from dynamic integrity constraints. Data and Knowledge Engineering 20(2), 163–193 (1996)
9. de Haan, L., Koppelaars, T.: Applied Mathematics for Database Professionals. Apress, USA (2007)
10. Hudnott, E., Sinclair, J., Darwen, H.: Rethinking database updates using a multiple assignment-based approach. WSEAS TRANSACTIONS on COMPUTERS 9(4), 392–405 (2010)
11. International Organization for Standardization: Foundation (SQL/Foundation). ISO/IEC 9075-2:2003 (2003)
12. Lipeck, U.W., Saake, G.: Monitoring dynamic integrity constraints based on temporal logic. Information Systems 12(3), 255–269 (1987)
13. Selzer, B.: RM Very Strong Suggestion 4: Transition constraints (2010, Sep 14), `http://www.dbmonster.com/Uwe/Forum.aspx/db-theory/2390/RM-VERY-STRONG-SUGGESTION-4-TRANSITION-CONSTRAINTS`. Also posted to The Third Manifesto e-mail list. Subscription at `ttm-subscribe@thethirdmanifesto.com`
14. Stonebraker, M., Held, G., Wong, E., Kreps, P.: The design and implementation of Ingres. ACM Transactions on Database Systems 1(3), 189–222 (1976)