Essay 20

Toward a Multilevel Secure Relational Data Model

Sushil Jajodia and Ravi S. Sandhu

A large number of databases in the Department of Defense, the intelligence community, and civilian government agencies contain data that are classified to have different security levels. All database users are also assigned security clearances. It is the responsibility of a multilevel secure (MLS) database management system (DBMS) to assure that each user gains access — directly or indirectly — to only those data for which he has proper clearance. Some private corporations also use security levels and clearances to ensure secrecy of sensitive information, although their procedures for assigning these are much less formal than in the government.

Most commercial DBMSs provide some form of data security by controlling modes of access privileges of users to data [GRIF76, RABI88]. These discretionary access controls (DAC) do not provide adequate mechanisms for preventing unauthorized disclosure of information. Therefore, commercial DBMSs providing only DAC are not suitable for use in multilevel environments. Multilevel systems require additional mechanisms for enforcing mandatory (or nondiscretionary) access controls (MAC) [DENN82].

As a result, there are several efforts under way to build multilevel secure relational DBMSs. These efforts are following the same path taken by object-oriented databases. On one hand, several database vendors (Oracle, Sybase, and Trudata, to name a few) are busy building commercial products, and others (for example, SRI [DENN87, LUNT90] and SCTC [HAIG91a]) are building research prototypes. On the other hand, there is no clear consensus regarding what exactly an MLS relational data model is. This has led to continuing arguments about basic principles such as integrity requirements and update semantics. This lack of consensus on fundamental issues underscores the subtleties involved in extending the classical relational model to a multilevel environment. In the absence of a strong theoretical framework it is unfortunate, but inevitable, that much

of the argument on basic issues is unduly influenced by implementation details of specific projects.

Our aim in this essay is to discuss the most fundamental aspects of the MLS relational model. It is our goal to be formal, analytical, and objective — in the sense of implementation independent. The contents of this essay are summarized in the following subsections.

Core integrity properties. It is important to specify precisely all constraints that relations must satisfy, since these constraints ensure that all instances in the database are meaningful. It is equally important to require only the minimal necessary constraints so as to allow as large a class of admissible instances as possible. In classical relational theory (see C.J. Date's work [DATE86], for example) the essential constraints have been identified as entity integrity and referential integrity. In a later section, we consider the multilevel analog of entity integrity. We identify four core integrity properties that should be required of all multilevel relations [JAJO91b]. One of these is a generalization of the usual entity integrity requirement to a multilevel context, while the other three are new to multilevel relations. For each property, we show why it is needed in multilevel relations. Our focus in this essay is on single relations, and we do not consider multilevel referential integrity here.

Relation updates. Somewhat paradoxically, the understanding of update operations is crucial to achieving secrecy of information in multilevel systems. We give a formal operational semantics for update operations on multilevel relations, that is, relations in which individual data elements are classified at different levels [JAJO90f]. For this purpose, the familiar INSERT, UPDATE, and DELETE operations of SQL [DATE86] are suitably generalized. Our goal here has been to preserve as much as possible the intuitive simplicity of these operations in classical relations without sacrificing security in the process. The main difference, with respect to the classical semantics of these operations, is that certain updates cannot be carried out by overwriting the data in place because doing so would result in leakage or destruction of secret information. This inescapable fact complicates the semantics of multilevel relations. These operations are consistent (or sound) in that all relations that can be constructed will satisfy the basic integrity properties required of multilevel relations. Additionally, these operations are complete in that every multilevel relation can be constructed by some sequence of these operations.

Decomposition and recovery of multilevel relations. We give a decomposition algorithm that breaks a multilevel relation into single-level relations and a new recovery algorithm that reconstructs the original multilevel relation from the decomposed single-level relations [JAJO91b]. There are several novel aspects to these decomposition and recovery algo-

rithms, which provide substantial advantages over previous proposals [DENN87, JAJO90c, LUNT90]:

1. These algorithms are formulated in the context of an operational semantics for multilevel relations, defined here by generalizing the usual UPDATE operations of SQL to multilevel relations.
2. These algorithms, with minor modifications, can easily accommodate alternative update semantics that have been proposed in the literature.
3. These algorithms are efficient because recovery is based solely on unionlike operations without any use of joins.
4. The decomposition is intuitively and theoretically simple, giving us a sound basis for correctness.

Overview. The rest of this essay is organized as follows. The next section gives an overview of basic concepts of multilevel security. Then we review basic definitions for standard (single-level) relations; those for multilevel relations follow. We offer four core integrity requirements (together with their justification) that we feel must be met by all multilevel relations. Then we examine various UPDATE operations in a multilevel context, as outlined above. Before concluding, we give the decomposition and recovery algorithms that have been formulated in terms of UPDATE operations defined in the previous section.

Basic relational concepts and security requirements

The standard relational model is concerned with data without security classifications. Data are stored in tables, called *relations*. Each relation has a number of columns, called *attributes*. At any given time, a relation contains a number of rows, called *tuples*. The number of tuples in a relation varies with time. As an example, consider the relation SOD given in Figure 1, which contains for each starship its name, its objective, and its destination.

Starship	Objective	Destination
Enterprise	Exploration	Talos
Voyager	Spying	Mars

Figure 1. SOD.

There is a *relation scheme* corresponding to each relation, consisting of the relation name together with a list of its attribute names. The relation scheme for the relation SOD is denoted as follows:

SOD(Starship, Objective, Destination)

While the scheme for a relation is invariant over time, a relation is not static over time. Tuples are continuously being inserted, deleted, or updated in a relation to reflect changes in the real world. Not all possible relations are meaningful in an application; only those that satisfy certain integrity constraints are considered valid.

Let $R(A_1, A_2, ..., A_n)$ be a relation scheme, and let X and Y denote sets of one or more of the attributes A_i in R. We say Y is *functionally dependent* on X, written $X \rightarrow Y$, if any relation for R satisfies at all times the following property: It does not have two tuples with the same values for X but different values for Y.

A *candidate key* of a relation scheme R is a minimal set of attributes on which all other attributes of R are functionally dependent. The *primary key* of a relation scheme R is one of its candidate keys that has been specifically designated as such.

Moving on to a multilevel world, a major issue is how access classes are assigned to data stored in relations. Access classes can be assigned to relations, to individual tuples in a relation, to individual attributes of a relation, or to individual data elements of the tuples of a relation. In this essay, we will consider the general (and most difficult) case, and assign access classes to individual data elements of a relation.

As a consequence of Bell-LaPadula restrictions, subjects having different clearances see different versions of a multilevel relation: A user having a clearance at an access class c sees only that data which lies at class c or below. As an example, consider the relation scheme SOD(Starship, Objective, Destination), where Starship is the primary key and the security classifications are assigned at the granularity of individual data elements. A user with Secret clearance will see the entire multilevel relation SOD_S shown in Figure 2, while a user having Unclassified clearance will see only the filtered relation SOD_U shown in Figure 3.

Now, consider once again the multilevel relation given in Figure 2. Suppose that a U-user who sees the instance in Figure 3 wishes to replace the second tuple of SOD_U by the tuple (Voyager, Exploration, Talos). From a purely database perspective, this update by the U-user should be rejected because the attribute Starship constitutes the primary key of SOD_S. However, from the security viewpoint, this update cannot be rejected since doing so will be sufficient to establish a downward signaling channel. Since a Secret process can send one bit of information by either inserting or deleting a particular tuple at the Secret level, both Secret and Unclassified processes can cooperate to establish a covert channel.

Thus, both tuples (Voyager, Spying, Mars) and (Voyager, Exploration, Talos) must somehow coexist in SOD_S, as in Figure 4. This is called *polyinstantiation:* There are two or more tuples in a multilevel relation with the same primary key.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Voyager	U	Spying	S	Mars	S	S

Figure 2. SOD_S.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Voyager	U	Null	U	Null	U	U

Figure 3. SOD_U.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Voyager	U	Exploration	U	Talos	U	U
Voyager	U	Spying	S	Mars	S	S

Figure 4. SOD_S.

Thus, we see that even the basic relational notion of a key does not have a straightforward extension to multilevel relations. Polyinstantiation illustrates the intrinsic difficulty of extending the standard relational concepts to the multilevel world; therefore, we devote a separate essay (Essay 21) to this problem. In this essay, our position is that there is a need for polyinstantiation in multilevel systems. However, it must be carefully controlled to avoid confusion and ambiguity in the database. For example, the S-instance of Figure 5 should not be allowed because it gives ambiguous information about the Voyager's objective at the S level.

Throughout this essay, we use the terms *high* and *low* to refer to two access classes such that the former is strictly higher than the latter in

the partial order. Also, if a user is logged on at an access class c, we refer to such a user as a c-user.

Starship		Objective		Destination		TC
Voyager	U	Exploration	S	Mars	S	S
Voyager	U	Spying	S	Mars	S	S

Figure 5. An illegal S-instance.

Multilevel relations

In this section, we review the basic concepts for the multilevel relations. In the next section, we will state four core integrity requirements that we feel must be satisfied by all multilevel relations. A *multilevel relation* consists of two parts: a relation scheme and relation instances.

Definition 1: Relation scheme. A state-invariant multilevel *relation scheme* is of the form

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

where each A_i is a *data attribute* over domain D_i, each C_i is a *classification attribute* for A_i, and TC is the *tuple-class attribute*. The domain of C_i is specified by a set $\{L_i, \ldots, H_i\}$ which enumerates the allowed values for access classes, ranging from the greatest lower bound (glb) L_i to the least upper bound (lub) H_i. The domain of TC is the set $\{\text{lub}\{L_i: i = 1, \ldots, n\}, \ldots, \text{lub}\{H_i: i = 1, \ldots, n\}\}$.

Definition 2: Relation instances. For each relation scheme, there is a collection of state-dependent *relation instances*

$$R_c(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

one for each access class c in the given lattice. Each relation instance is a set of distinct tuples of the form $(a_1, c_1, a_2, c_2, \ldots, a_n, c_n, tc)$, where each $a_i \in D_i$ or a_i = null, $c \geq c_i$, and $tc = \{\text{lub}\{c_i: i = 1, \ldots, n\}$. Moreover, if a_i is not null, then $c_i \in \{L_i, \ldots, H_i\}$. We require that c_i be defined even if a_i is null — that is, a classification attribute cannot be null.

The multiple relation instances are, of course, related; each instance is intended to represent the version of reality appropriate for each access class. Roughly speaking, each element $t[A_i]$ in a tuple t is visible in instances at access class $t[C_i]$ or higher; $t[A_i]$ is replaced by a null value in

an instance at a lower access class. We give a more formal description using the filter function in the next section.

Core integrity properties

In this section, we state four core integrity properties that must be satisfied by all multilevel relations. For each property, we justify why it is necessary.

Since a multilevel relation has different instances at different access classes, it is inherently more complex than a standard relation. In a standard relation, the definition of keys is based on functional dependencies. In a multilevel setting, the concept of functional dependencies is itself clouded because a relation instance is now a collection of sets of tuples rather than a single set of tuples.

We assume that there is a user-specified primary key *AK* consisting of a subset of the data attributes A_i. This is called the *apparent primary key* of the multilevel relation scheme. We will return to the issue of what constitutes the primary key of a multilevel relation after we define the polyinstantiation integrity property.

In general, *AK* will consist of multiple attributes. Entity integrity from the standard relational model prohibits null values for any of the attributes in *AK*. This property [DENN87] extends to multilevel relations, as shown in the following subsections.

Property 1: Entity integrity. Let *AK* be the apparent key of *R*. A multilevel relation *R* satisfies entity integrity if and only if for all instances R_c of *R* and $t \in R_c$:

1. $A_i \in AK \Rightarrow [A_i] \neq \text{null}$;
2. $A_i, A_j \in AK \Rightarrow t[C_i] = t[C_j]$, that is, *AK* is uniformly classified; and
3. $A_i \notin AK \Rightarrow t[C_i] \geq t[C_{AK}]$ (where C_{AK} is defined to be the classification of the apparent key).

The first requirement is an obvious carryover from the standard relational model and ensures that no tuple in R_c has a null value for any attribute in *AK*. The second requirement says that all *AK* attributes have the same classification in a tuple, that is, they are either all U or all S, and so on. This will ensure that *AK* is either entirely visible or entirely null at a specific access class *c*. The third requirement states that in any tuple the class of the non-*AK* attributes must dominate C_{AK}. This rules out the possibility of associating nonnull attributes with a null primary key.

At this point it is important to clarify the semantics of null values. There are two major issues:

1. the classification of null values, and
2. the subsumption of null values by nonnull ones.

Our requirements are respectively that null values be classified at the level of the key in the tuple, and that a null value is subsumed by a nonnull value independent of the latter's classification. These two requirements are formally stated in Property 2.

Property 2: Null integrity. A multilevel relation R satisfies null integrity if and only if for each instance R_c of R both of the following conditions are true:

1. For all $t \in R_c$, $t[A_i] = \text{null} \Rightarrow t[C_i] = t[C_{AK}]$; that is, nulls are classified at the level of the key.
2. Let us say that tuple t subsumes tuple s if for every attribute A_i, either (a) $t[A_i, C_i] = s[A_i, C_i]$ or (b) $t[A_i] \neq \text{null}$ and $s[A_i] = \text{null}$. Our second requirement is that R_c is subsumption free in the sense that it does not contain two distinct tuples such that one subsumes the other.

We will henceforth assume that all computed relations are made subsumption free by exhaustive elimination of subsumed tuples. The null integrity requirement was identified in an earlier work [JAJO90c].

Consider the relation instance for SOD given in Figure 6. The motivation behind the null integrity property is that if an S-user updates the destination of Enterprise to be Rigel, he or she will see the instance given in Figure 7 rather than the one given in Figure 8, since the first tuple in Figure 8 is subsumed by the second tuple.

The next property is concerned with consistency between the different relation instances. The need for such a property was identified earlier [DENN87], but the formulations were incorrect. The correct formulation [JAJO90c] was adopted by SeaView researchers [LUNT90].

Property 3: Interinstance integrity. R satisfies interinstance integrity if and only if for all $c' \leq c$ we have $R_{c'} = \sigma(R_c, c')$, where the *filter function* σ produces the c'-instance $R_{c'}$ from R_c as follows:

1. For every tuple $t \in R_c$ such that $t[C_{AK}] \leq c'$, there is a tuple $t' \in R_{c'}$ with $t'[AK, C_{AK}] = t[AK, C_{AK}]$ and for $A_i \notin AK$

$$t'[A_i, C_i] = \begin{cases} t[A_i, C_i] & \text{if } t[C_i] \leq c' \\ <\text{null}, t[C_{AK}]> & \text{otherwise} \end{cases}$$

2. There are no tuples in $R_{c'}$ other than those derived by the above rule.

3. The end result is made subsumption free by exhaustive elimination of subsumed tuples.

The filter function maps a multilevel relation to different instances, one for each descending access class in the security lattice. Filtering limits each user to that portion of the multilevel relation for which he or she is cleared. Thus, for example, an S-user will see the entire relation given in Figure 7, while a U-user will see the filtered instance given in Figure 6. It is evident that $\sigma(R_c, c) = R_c$, and $\sigma(\sigma(R_c, c'), c'') = \sigma(R_c, c'')$ for $c \geq c' \geq c''$, as one would expect from the intuitive notion of filtering.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Null	U	U

Figure 6. SOD_U.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Rigel	S	S

Figure 7. SOD_S.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Null	U	U
Enterprise	U	Exploration	U	Rigel	S	S

Figure 8. Violation of null integrity.

We are now ready to state our fourth and final property. In a standard relation there cannot be two tuples with the same primary key. In a multilevel relation we will similarly expect that there cannot be two tuples with the same *apparent primary key*. However, as we observed earlier, secrecy considerations compel us to allow multiple tuples with the same apparent primary key. (See, however, Essay 21 on polyinstantiation.) We have the following property to control the manner in which this can be done.

Property 4: Polyinstantiation integrity. R satisfies polyinstantiation integrity (PI) if and only if for every R_c we have for all A_i: AK, C_{AK}, $C_i \rightarrow A_i$.

This property stipulates that the user-specified apparent key AK, in conjunction with the classification attributes C_{AK} and C_i, functionally determines the value of the A_i attribute. Thus, PI allows the instance in Figure 4 while ruling out the S-instance of Figure 5.

Property 4 implicitly defines what is meant by the primary key in a multilevel relation. The primary key of a multilevel relation is $AK \cup C_{AK} \cup C_R$ (where AK is the set of data attributes constituting the user-specified primary key, C_{AK} is the classification attribute for data attributes in AK, and C_R is the set of classification attributes for data attributes not in AK). This is because from PI it follows that the functional dependency $AK \cup C_{AK} \cup C_R \rightarrow A_R$ holds (where A_R denotes the set of all attributes that are not in AK). Note that for single-level relations, C_{AK} and C_R will be equal to the same constant value in all tuples. Therefore, in this case, PI amounts to saying that $AK \rightarrow A_R$, which is precisely the definition of the primary key in relational theory.

When Property 4 was originally proposed [DENN87], it was coupled with an additional multivalued dependency[1] (MVD) requirement AK, $C_{AK} \rightarrow\rightarrow A_i$, C_i to be satisfied by every instance. There are unpleasant consequences of this multivalued dependency [JAJO90c]. Hence, our position is that polyinstantiation integrity should require only the functional dependency stated in Property 4.

The UPDATE operations

In this section, we discuss in detail the three UPDATE operations (INSERT, UPDATE, and DELETE). We keep the syntax for these operations identical to the standard SQL.

Let $R(A_1, C_1, \ldots, A_n, C_n, TC)$ be a multilevel relation scheme. To simplify the notation, we use A_1 instead of AK to denote the apparent primary key.

Consider a user logged on at access class c. Now a c-user directly sees and interacts with the c-instance R_c. From the viewpoint of this user, the remaining instances of R can be categorized into three cases: Those strictly dominated by c, those that strictly dominate c, and those incomparable with c. The following notation is useful for ease of reference to these three cases:

1. $R_{c' < c} \equiv R_{c'}$, such that $c' < c$.
2. $R_{c' > c} \equiv R_{c'}$, such that $c' > c$.
3. $R_{c' \sim c} \equiv R_{c'}$, such that c' is incomparable with c.

[1] See [DATE86] for a definition of multivalued dependency.

Security considerations, and in particular the *-property, dictate that a *c*-user cannot insert, update, or delete a tuple, directly or indirectly (as a side effect) in any $R_{c' < c}$ or $R_{c' \sim c}$. Since actions of a *c*-user cannot have an impact on any $R_{c' < c}$, the effect of insertion, update, or deletion must be confined to those tuples in R_c with tuple class equal to *c*. Because of the interinstance property, these changes must be properly reflected in the instances $R_{c' > c}$. The latter effect is only partly determined by the core integrity properties presented earlier, leaving room for different interpretations [HAIG91a, JAJO90c, JAJO90f, LUNT91, SAND90a].

Strictly speaking, in all cases we should speak of operations being performed by a *c*-subject (or *c*-process) rather than a *c*-user. It is, however, easier to intuitively consider the semantics by visualizing a human being interactively carrying out these operations. The semantics do apply equally well to processes operating on behalf of a user, whether interactive or not.

The INSERT statement. The INSERT statement executed by a *c*-user has the following general form, where the *c* is implicitly determined by the user's login class:

```
INSERT
INTO      R_c[(A_i [, A_j]...)]
VALUES    (a_i [, a_j]...)
```

In this notation, the brackets denote optional items and the ellipsis (...) signifies repetition. If the list of attributes is omitted, it is assumed that all the data attributes in R_c are specified. Only data attributes A_i can be explicitly given values. The classification attributes C_i are all implicitly given the value *c*.

Let t be the tuple such that $t[A_k] = a_k$ if A_k is included in the attributes list in the INSERT statement, $t[A_k]$ = Null if A_k is not in the list, and $t[C_l] = c$ for $1 \le l \le n$. The insertion is permitted if and only if:

1. $t[A_1]$ does not contain any nulls, and
2. for all $u \in R_c$: $u[A_1] \neq t[A_1]$.

If so, the tuple t is inserted into R_c and by side effect into all $R_{c' > c}$. This is, moreover, the only side effect visible in any $R_{c' > c}$.

To illustrate, suppose a U-user wishes to insert a second tuple into the SOD instance given in Figure 9. He or she does so by executing the following INSERT statement:

```
INSERT
INTO      SOD
VALUES    ('Voyager', 'Exploration', 'Mars')
```

As a result of the INSERT statement, the U-instance of SOD will become as shown in Figure 10. This insertion is straightforward and identical to what happens in single-level relations.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U

Figure 9. $SOD_U = SOD_S$.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Voyager	U	Exploration	U	Mars	U	U

Figure 10. SOD_U.

On the other hand, suppose an S-user wishes to insert the following tuple into the SOD instance of Figure 9:

```
INSERT
INTO      SOD
VALUES    ('Enterprise', 'Spying', 'Rigel')
```

In this case, we can either reject the insert or accept it and allow two tuples with the same apparent key Enterprise to coexist, as shown in Figure 11. The two tuples in Figure 11 are regarded as pertaining to two distinct entities. We call such situations *optional polyinstantiations*. Insertion of the secret tuple is not required for closing signaling channels. It is secure to reject such insertions.

Finally, we illustrate the situation where polyinstantiation is required to close signaling channels. Consider the SOD_S instance given in Figure 12. U-users see an empty instance SOD_U. Suppose a U-user executes the following INSERT statement:

```
INSERT
INTO      SOD
VALUES    ('Enterprise', 'Exploration', 'Talos')
```

This insertion cannot be rejected on the grounds that a tuple with apparent key Enterprise has previously been inserted by an S-user. Doing so

would establish a signaling channel from S to U. Therefore, by security considerations we are compelled to allow insertion of this tuple. In such cases, we say we have *required polyinstantiation*. The effect of this insertion by a U-user is to change SOD_S from Figure 12 to Figure 11.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	S	Spying	S	Rigel	S	S

Figure 11. SOD_S.

Starship		Objective		Destination		TC
Enterprise	S	Spying	S	Rigel	S	S

Figure 12. SOD_S.

The UPDATE statement. Our interpretation of the semantics of an update command is close to the one in the standard relational model: An update command is used to change values in tuples that are already present in a relation. UPDATE is a set-level operator; that is, all tuples in the relation which satisfy the predicate in the UPDATE statement are to be updated (provided the resulting relation satisfies polyinstantiation integrity). Since we are dealing with multilevel relations, we may have to polyinstantiate some tuples. However, addition of tuples due to polyinstantiation is to be minimized to the extent possible.

The UPDATE statement executed by a *c*-user has the following general form:

UPDATE R_c
SET $A_i = s_i$ [, $A_j = s_j$] ...
[WHERE p]

Here s_k is a scalar expression, and p is a predicate expression which identifies those tuples in R_c that are to be modified. The predicate p may include conditions involving the classification attributes, in addition to the usual case of data attributes. The assignments in the SET clause, however, can involve only the data attributes. The corresponding classification attributes are implicitly determined to be *c*.

The intent of the UPDATE operation is to modify $t[A_k]$ to s_k in those tuples t in R_c that satisfy the given predicate p. In multilevel relations, however, we have to implement the intent slightly differently to prevent illegal information flows.

Examples of UPDATE operations. Consider the SOD instances given in Figures 13 and 14. Suppose the U-user makes the following update to SOD_U, shown in Figure 13:

```
UPDATE  SOD
SET     Destination = Talos
WHERE   Starship = 'Enterprise'
```

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Null	U	U

Figure 13. SOD_U.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Rigel	S	S

Figure 14. SOD_S.

The changes to SOD_U in Figure 13 and SOD_S in Figure 14 are shown in Figures 15 and 16, respectively. Note that in SOD_S the Destination attribute for the Enterprise is now polyinstantiated. This is an example of required polyinstantiation that cannot be completely eliminated without introducing signaling channels or limiting the expressive capability of the database.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U

Figure 15. SOD_U.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	U	Exploration	U	Rigel	S	S

Figure 16. SOD_S.

Next, suppose starting with the instance SOD_S of Figure 16 an S-user invokes the following update:

```
UPDATE  SOD
SET     Objective = Spying
WHERE   Starship = 'Enterprise' AND
        Destination = 'Rigel'
```

In this case, SOD_S will change to the instance given in Figure 17, not to the instance given in Figure 18. That is, the update is interpreted as applying only to the second tuple in Figure 16, not to the first tuple. The S-user can go from Figure 16 to Figure 18 by issuing the following update:

```
UPDATE  SOD
SET     Objective = Spying
WHERE   Starship = 'Enterprise'
```

This update is interpreted as applying to both tuples of Figure 16. The first two tuples of Figure 18 result from polyinstantiation of the first tuple of Figure 16. The third tuple of Figure 18 results from the normal replacement update of the second tuple of Figure 16.

Next, suppose a U-user makes the following update to the relation shown in Figure 15 (assume S-users see the instance given in Figure 16):

```
UPDATE  SOD
SET     Objective = Spying
WHERE   Starship = 'Enterprise'
```

As a consequence of the above update, not only SOD_U will change from the relation in Figure 15 to the one in Figure 19, but SOD_S will also change from the relation in Figure 16 to the one in Figure 20. Thus, polyinstantiation integrity is preserved in instances at different security levels. Note in particular how the secret tuple in Figure 16 has changed to the secret tuple in Figure 20 due to an update by a U-user.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	U	Spying	S	Rigel	S	S

Figure 17. SOD_S.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	U	Spying	S	Talos	U	S
Enterprise	U	Spying	S	Rigel	S	S

Figure 18. SOD_S.

Starship		Objective		Destination		TC
Enterprise	U	Spying	U	Talos	U	U

Figure 19. SOD_U.

Starship		Objective		Destination		TC
Enterprise	U	Spying	U	Talos	U	U
Enterprise	U	Spying	U	Rigel	S	S

Figure 20. SOD_S.

Effect at the user's access class. We now formalize and further develop the ideas sketched out above. First consider the effect of an UPDATE operation by a c-user on R_c. Let

$$S = \{t \in R_c\colon\ t \text{ satisfies the predicate } p\}$$

We describe the effect of the UPDATE operation by considering each tuple $t \in S$ in turn. The net effect is obtained as the cumulative effect of updating each tuple in turn. The UPDATE operation will succeed if and only if at every step in this process polyinstantiation integrity is maintained.

Otherwise, the entire UPDATE operation is rejected and no tuples are changed. In other words, UPDATE has an all-or-nothing integrity failure semantics.

It remains to consider the effect of UPDATE on each tuple $t \in S$. There are two components to this effect. First, tuple t is replaced by tuple t', which is identical to t except for those data attributes that are assigned new values in the SET clause. This is the familiar replacement semantics of UPDATE in a single-level world. In terms of our earlier examples, the update of SOD_U from Figure 13 to Figure 15 and then to Figure 19 illustrates this semantics. The formal definition of the tuple t' obtained by replacement semantics is straightforward, as follows:

$$t'[A_k, C_k] = \begin{cases} t[A_k, C_k] & A_k \notin \text{SET clause} \\ <s_k, c> & A_k \in \text{SET clause} \end{cases}$$

Second, to avoid signaling channels, we may need to introduce an additional tuple t'' to hide the effects of the replacement of t by t' from users at levels below c (c is the level of the user executing the UPDATE). This will occur whenever there is some attribute A_k in the SET clause with $t[C_k] < c$. The idea is that the original value of $t[A_k]$ with classification $t[C_k]$ is preserved in t''. At the same time, the core integrity properties presented earlier must be preserved.

To be concrete, consider our earlier example of the update of SOD_S from Figure 16 to Figure 17. The WHERE clause of the UPDATE statement picks up the second tuple in Figure 16, which by replacement semantics gives us the second tuple in Figure 17. In this case, the unclassified Exploration value of the Objective attribute continues to be available in the first tuple of Figure 17, and we need not introduce an additional tuple to hide the effect of this update from U-users. On the other hand, suppose the same UPDATE statement, that is,

```
UPDATE  SOD
SET     Objective = Spying
WHERE   Starship = 'Enterprise' AND
        Destination = 'Rigel'
```

was executed by an S-user in the context of Figure 14. Prior to the update, U-users see the instance in Figure 13 and therefore must continue to do so after the update. To achieve this, SOD_S changes from Figure 14 to Figure 21. The first tuple in Figure 21 is the tuple t' dictated by the usual replacement semantics. The second tuple is the t'' tuple introduced to hide the effect of the update from U-users and maintain interinstance integrity. It should be noted that Figure 22 also achieves these two goals.

However, it does so at the cost of a spurious association between Rigel and Exploration, which is avoided in Figure 21.

Starship		Objective		Destination		TC
Enterprise	U	Spying	S	Rigel	S	S
Enterprise	U	Exploration	U	Null	U	U

Figure 21. SOD_S.

Starship		Objective		Destination		TC
Enterprise	U	Spying	S	Rigel	S	S
Enterprise	U	Exploration	U	Rigel	S	S

Figure 22. SOD_S.

We now give a formal definition of the t'' tuple introduced to close the signaling channel:

$$t''[A_k, C_k] = \begin{cases} t[A_k, C_k] & t[C_k] < c \\ <\text{Null}, t[A_1]> & t[C_k] = c \end{cases}$$

To summarize, each tuple $t \in S$ is replaced by t' and possibly in addition by t'' (if t'' exists). The update is successful if the resulting relation satisfies polyinstantiation integrity. Otherwise, the update is rejected, and the original relation is left unchanged.

Effect above the user's access class. Next, consider the effect of the UPDATE operation on $R_{c'>c}$. This, of course, assumes that the UPDATE operation on R_c was successful. Unfortunately, the core integrity properties do not uniquely determine how an update by a c-user to R_c should be reflected in updates to $R_{c'>c}$. Several different options have been proposed [HAIG91a, JAJO90f, LUNT90, LUNT91]. In this section, we will adopt the *minimal propagation rule* [JAJO90f]. This rule introduces exactly those tuples in $R_{c'>c}$ needed to preserve the interinstance property — that is, put t' and t'' (if t'' exists and survives subsumption) in each $R_{c'>c}$ and nothing else.[2]

[2]The minimal propagation rule needs to be slightly extended to achieve completeness (that is, every multilevel relation can be constructed by some sequence of update operations).

Formally, the effect of the UPDATE operation is again best explained by focusing on a particular tuple t in S. Let A_k be an attribute in the SET clause such that:

1. $t[C_k] = c$ and
2. $t[A_k] = x$, where x is nonnull.

That is, the c-user is actually changing a nonnull value of $t[A_k]$ at his own level to s_k. Now consider $R_{c' > c}$. Due to polyinstantiation, there may be several tuples u in $R_{c' > c}$ which have the same apparent primary key as t (that is, $u[A_1, C_1] = t[A_1, C_1]$) and match t in the A_k and C_k attributes (that is, $u[A_k, C_k] = t[A_k, C_k]$). To maintain polyinstantiation integrity (that is, Property 4 presented earlier), we must therefore change the value of $u[A_k]$ from x to s_k. This requirement is formally stated as follows:

1. For every $A_k \in$ SET clause with $t[A_k] \neq$ Null, let

   $$U = \{u \in R_{c' > c}: u[A_1, C_1] = t[A_1, C_1] \wedge u[A_k, C_k] = t[A_k, C_k]\}$$

   Polyinstantiation integrity dictates that we replace every $u \in U$ by u' identical to u, except for

   $$u'[A_k, C_k] = <s_k, c>$$

   This rule applies cumulatively for different A_k's in the SET clause. This requirement is absolute and must be rigidly enforced by the DBMS.

2. The second requirement is imposed by the interinstance integrity property. To maintain interinstance integrity, we insert t' and t'' (if it exists and survives subsumption) in $R_{c' > c}$.

The second requirement is weaker than the first, in that interinstance integrity only stipulates what minimum action is required. We can insert a number of additional tuples v in $R_{c' > c}$ with $v[A_1, C_1] = t'[A_1, C_1]$, so long as the core integrity properties are not violated. In particular, if t' subsumes the tuple in $\sigma(\{v\}, c)$, interinstance integrity is still maintained. Minimal propagation makes the simplest assumption in this case; that is, only t' and t'' are inserted in $R_{c' > c}$, and nothing else is done.

The DELETE statement. The DELETE statement has the following general form:

```
DELETE
FROM    Rc
[WHERE  p ]
```

Here, p is a predicate expression which helps identify those tuples in R_c that are to be deleted. The intent of the DELETE operation is to delete those tuples t in R_c that satisfy the given predicate. But in view of the *-property, only those tuples t that additionally satisfy $t[TC] = c$ are deleted from R_c. To maintain interinstance integrity, polyinstantiated tuples are also deleted from $R_{c' > c}$.

In particular, if $t[C_1] = c$, then any polyinstantiated tuples in $R_{c' > c}$ will be deleted from $R_{c' > c}$. Hence, the entity that t represents will completely disappear from the multilevel relation. On the other hand, with $t[C_1] < c$, the entity will continue to exist in $R_{t[C_1]}$ and in $R_{c' > t[C_1]}$.

Decomposition and recovery

In this section, we give the decomposition and recovery algorithms [JAJO91b] formulated in terms of UPDATE operations defined in the previous section. We give an abstract description and complete formal statement first and defer consideration of examples until later.

Background. In multilevel relational DBMSs, a major issue is how access classes are assigned to data stored in relations. The proposals have included assigning access classes to relations [GROH76], assigning access classes to individual tuples in a relation [GARV86], and assigning access classes to individual attributes of a relation [HINK75]. Unlike these proposals, in the Secure Data Views (SeaView) project, security classifications are assigned to individual data elements of the tuples of a relation [DENN87, DENN88a, LUNT90] (for example, see Figure 23).

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Mine	C	Sirius	C	C
Ent	U	Spy	S	Rigel	S	S
Ent	U	Coup	TS	Orion	TS	TS

Figure 23. A multilevel relation SOD.

Multilevel relations in SeaView exist only at the logical level. In reality, multilevel relations are decomposed into a collection of single-level base relations that are then physically stored in the database. Completely transparent to users, multilevel relations are reconstructed from these base relations on user demand. There are several practical advantages of

being able to decompose and store a multilevel relation as a collection of single-level base relations. In particular, the underlying trusted computing base (TCB) can enforce mandatory controls with respect to the single-level base relations. This allows the DBMS to run mostly as an untrusted application with respect to the underlying TCB.

In SeaView, the decomposition of multilevel relations into single-level ones is performed by applying two different types of fragmentation: *horizontal* and *vertical*. Thus the multilevel relation in Figure 23 will be stored as *nine* single-level fragments (one primary key group relation and eight attribute group relations), shown in Figure 24. This leads to many problems with the SeaView decomposition and recovery algorithms:

1. *Repeated joins.* The vertical fragmentation used in SeaView results in single-level relations that consist of the key attribute, a single nonkey attribute, and their classification attributes. This means that nearly all queries involving multiple attributes necessitate repeated (left outer) joins of several single-level relations. It is well known that join is an expensive operation and should be avoided whenever possible [SCHK82].
2. *Spurious tuples.* Whenever a relation is stored as one or more fragments, it must be possible to reconstruct the original relation exactly from the fragments. This, however, is not the case with the SeaView decomposition. When the SeaView recovery algorithm is applied to the single-level relations in Figure 24, a Top Secret user will be shown the relation given in Figure 25. While the original Top Secret instance in Figure 23 describes four missions for the Enterprise, a Top Secret user will see the 16 missions of Figure 25 using the SeaView approach.
3. *Incompleteness.* The SeaView decomposition puts severe limitations on the expressive capability of the database. Several instances that have realistic and useful interpretations cannot be realized in SeaView [JAJO90c, JAJO91a].
4. *Left outer joins.* The SeaView recovery algorithm is based on the left outer join of relations. Many theoretical complications and pitfalls arise with outer joins [DATE86].

Elsewhere [JAJO90c] we have given a modified version of the SeaView decomposition and recovery algorithms. Our principal motivation was to give a lossless decomposition that avoids the spurious tuples introduced by SeaView. To achieve this, we store the relation in Figure 23 as a collection of 12 single-level relations. Figure 26 shows the four primary key group relations; the eight attribute group relations are identical to those given in Figure 24b. The recovery algorithm, when applied to these single-level base relations, yields exactly the original-instance SOD in Figure 23. While these algorithms eliminate the last three problems, the first

problem remains: Satisfying queries involving multiple attributes requires taking repeated natural joins of several single-level relations.

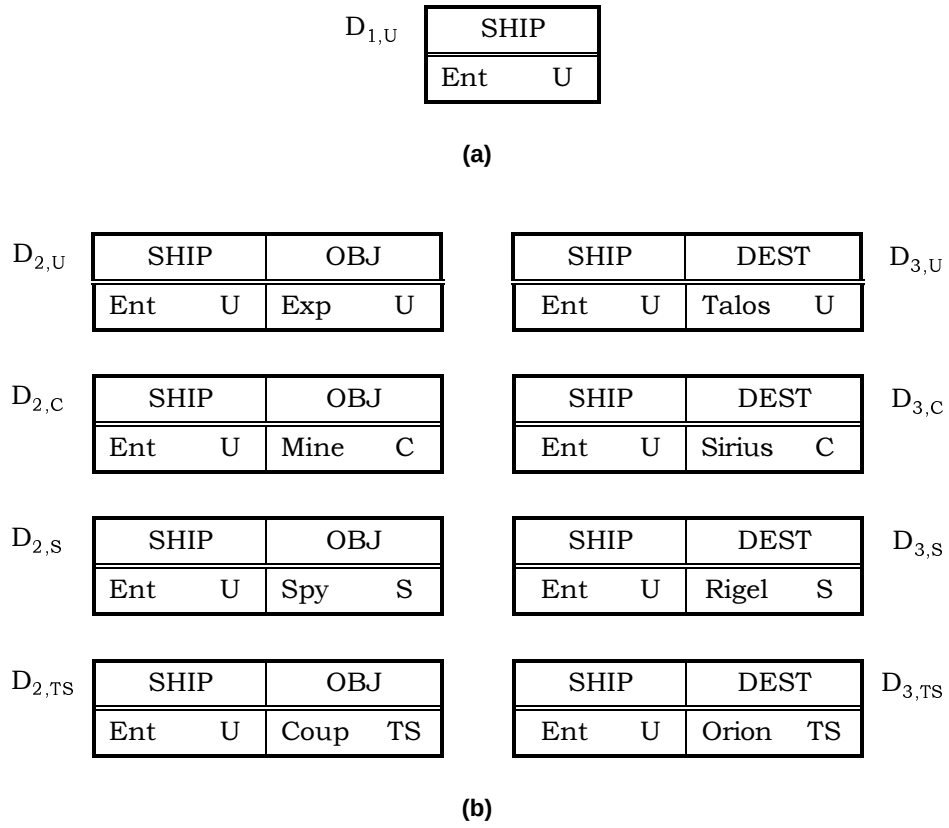

Figure 24. SeaView decomposition of Figure 23 into nine single-level base relations: (a) primary key group relation, (b) attribute group relations.

In this section, we give a decomposition algorithm and a recovery algorithm that have several advantages over the SeaView algorithms and our earlier algorithms discussed above [LUNT90, JAJO90c]:

1. The decomposition and recovery algorithms given below are based on operational semantics for the UPDATE operations on multilevel relations. The semantics of multilevel relations are defined here by generalizing the usual UPDATE operations of SQL.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Sirius	C	C
Ent	U	Mine	C	Talos	U	C
Ent	U	Mine	C	Sirius	C	C
Ent	U	Exp	U	Rigel	S	S
Ent	U	Mine	C	Rigel	S	S
Ent	U	Spy	S	Talos	U	S
Ent	U	Spy	S	Sirius	C	S
Ent	U	Spy	S	Rigel	S	S
Ent	U	Exp	U	Orion	TS	TS
Ent	U	Mine	C	Orion	TS	TS
Ent	U	Spy	S	Orion	TS	TS
Ent	U	Coup	TS	Talos	U	TS
Ent	U	Coup	TS	Sirius	C	TS
Ent	U	Coup	TS	Rigel	S	TS
Ent	U	Coup	TS	Orion	TS	TS

Figure 25. SeaView recovery algorithm applied to Figure 24.

2. These algorithms, with minor modifications, can easily accommodate alternative update semantics that have been proposed in the literature. It is even possible to keep the decomposition fixed and vary the recovery algorithms to realize these alternate semantics.
3. These algorithms are computationally efficient because the decomposition uses only horizontal fragmentation to break multilevel relations into single-level ones. The decomposition for the relation in Figure 23 is shown in Figure 27. Since the decomposition does not require any vertical fragmentation, it is possible to reconstruct a multilevel relation from the underlying single-level base relations without having to perform any (left or natural) joins; only unions are required.
4. The recovery and decompositions are simple to state and prove correct.

$D_{1,U}$

SHIP		C_2	C_3
Ent	U	U	U

$D_{1,C}$

SHIP		C_2	C_3
Ent	U	C	C

$D_{1,S}$

SHIP		C_2	C_3
Ent	U	S	S

$D_{1,TS}$

SHIP		C_2	C_3
Ent	U	TS	TS

Figure 26. Primary key group relations after Jajodia-Sandhu decomposition of Figure 23 into 12 single-level base relations. The eight attribute group relations are identical to those in Figure 24b.

This section is organized as follows. First we give the decomposition and recovery algorithms that preserve the update semantics proposed earlier. Then we give several examples to illustrate the behavior of the update semantics, as well as the decomposition and recovery algorithms. We also show how these algorithms, with minor modifications, can easily accommodate alternative update semantics proposed in the literature.

Decomposition. The decomposition has for each multilevel relation scheme

$$R(A_1, C_1, \ldots, A_n, C_n, TC)$$

a collection of single-level base relations

$$D_c(A_1, C_1, \ldots, A_n, C_n)$$

one for each access class c in the security class lattice. This is in contrast to the SeaView decomposition [LUNT90] and the Jajodia-Sandhu decomposition [JAJO90c], both of which require several single-level relations at each access class (compare Figures 24 and 26 with Figure 27).

A c-user always sees and interacts with the c-instance R_c. Whenever a c-user issues an insert, update, or delete command against R_c, tuples are

added, modified, or removed from the underlying base relation D_c. Any change in R_c must be properly reflected in $R_{c'>c}$ (and in $D_{c'>c}$), but this is accomplished during the recovery of a $R_{c'>c}$. Thus, when D_c is modified as the result of an update by a c-user, there are no changes made to any other $D_{c'}$, $c' \neq c$. Changes in $R_{c'>c}$ due to updates by c-users are accounted for by the recovery algorithm, which uses $\bigcup_{c' \leq c} D_{c'}$ to reconstruct an $R_{c'>c}$.

D_U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

D_C

SHIP		OBJ		DEST	
Ent	U	Mine	C	Sirius	C

D_S

SHIP		OBJ		DEST	
Ent	U	Spy	S	Rigel	S

D_{TS}

SHIP		OBJ		DEST	
Ent	U	Coup	TS	Orion	TS

Figure 27. New decomposition of Figure 23 into four single-level base relations.

The INSERT statement. Suppose as a result of the INSERT statement, a c-user successfully inserts the following tuple t in R_c: $t[A_k] = a_k$ if A_k is included in the attributes list in the INSERT statement, $t[A_k]$ = Null if A_k is not in the list, and $t[C_l] = c$ for $1 \leq l \leq n$. In this case, the decomposition will also insert the tuple t into D_c.

There are no other insertions. The recovery algorithm will use $\bigcup_{c' \leq c} D_{c'}$ to reconstruct an $R_{c'>c}$, and since t is in D_c, it will be in $R_{c'>c}$ as well.

The UPDATE statement. We next consider the effect of an UPDATE operation by a c-user on R_c. As we indicated earlier, only D_c will be modified by the decomposition algorithm.

Suppose that a c-user successfully executes the UPDATE statement presented earlier. Once again, let

$$S = \{t \in R_c\text{: } t \text{ satisfies the predicate } p\}$$

For each $t \in S$, there are two cases to consider:

1. $t[A_1, C_1] = c$. In this case, there can be no polyinstantiation of tuple t at the c level. There is exactly one tuple $u \in D_c$ with $u[A_1, C_1] = t[A_1, C_1]$. We replace u by the following tuple u': $u'[A_1, C_1] = u[A_1, C_1]$, and for $k \neq 1$,

$$u'[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ u[A_k, C_k] & A_k \notin \text{SET clause} \end{cases}$$

   Note that in this case $u'[C_l] = c$ for $1 \leq l \leq n$.

2. $t[A_1, C_1] < c$. In this case, tuple t will be polyinstantiated at the c level. There are two separate subcases, depending on whether or not t has been polyinstantiated at level c prior to the update. These subcases are as follows:

   (a) t is not polyinstantiated at level c prior to the update. In this case, there does not exist a tuple $u \in D_c$ with $u[A_1, C_1] = t[A_1, C_1]$. (Note that the tuple class of t must be strictly less than c.)
   We add a tuple u to D_c, where u is defined as follows: $u[A_1, C_1] = t[A_1, C_1]$, and for $k \neq 1$,

$$u[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ <?, t[C_k]> & A_k \notin \text{SET clause} \end{cases}$$

   The symbol "?" is a special symbol that can never be an actual value for an attribute. It plays an important role during recovery, as we will see in a moment. Informally, a "?" means that this value is to be obtained from the corresponding tuple in

$$D_{t[C_k]}$$

   (b) t is polyinstantiated at level c prior to the update. In this case, there will be one or more tuples $u \in D_c$ that satisfy the condition $u[A_1, C_1] = t[A_1, C_1]$, and for $k \neq 1$, (i) if $t[C_i] = c$, then $u[A_i, C_i] = t[A_i, C_i]$ and (ii) if $t[C_i] < c$, then $u[A_i, C_i] = <?, t[C_i]>$. For each tuple u that satisfies this condition, we replace u by the following tuple u': $u'[A_1, C_1] = u[A_1, C_1]$, and for $k \neq 1$,

$$u'[A_k, C_k] = \begin{cases} <s_k, c> & A_k \in \text{SET clause} \\ u[A_k, C_k] & A_k \notin \text{SET clause} \end{cases}$$

The DELETE statement. Finally, suppose a c-user executes the DELETE statement given earlier, and as a result all tuples t that satisfy the predicate p and $t[TC] = c$ are deleted from R_c. In terms of the decomposition, for each such t, we delete from D_c the tuple u that satisfies the following condition: $u[A_1, C_1] = t[A_1, C_1]$, and for $k \neq 1$, (i) if $t[C_i] = c$, then $u[A_i, C_i] = t[A_i, C_i]$ and (ii) if $t[C_i] < c$, then $u[A_i, C_i] = <?, t[C_i]>$.

Summary. To summarize, whenever a c-user updates the instance R_c, all changes are confined to the underlying base relation D_c. These changes leave ripple marks on $R_{c' > c}$, but this is accomplished when an $R_{c' > c}$ is constructed using the recovery algorithm described next.

Recovery algorithm. We are now prepared to give the recovery algorithm. To recover the instance R_c at an access class c, the following steps are taken:

1. Form the union $\bigcup_{c' \leq c} D_{c'}$. Extend each tuple t in the result by appending to it its tuple class computed as $t[TC] = \text{lub}\{t[C_i]: i = 1, ..., n\}$. Call the end result R_c.

2. Next, apply the following *key deletion rule* to R_c:

   Let $t_1 \in R_c$ be such that $t_1[C_1] < c$ and R_c does not contain a t_2 such that $t_2[A_1, C_1] = t_1[A_1, C_1]$ and $t_2[TC] = t_1[C_1]$. Then we delete t_1 from R_c. If $t_1[TC] = c$, then we delete t_1 from D_c as well.
   (Comment: The motivation for the key deletion rule is that a low user has deleted the tuple key. We therefore delete all higher tuples with that low key as well. Clearly t_1 is no longer needed, and its elimination amounts to garbage collection. We could alternately place tuples such as t_1 in a separate relation and have them examined by a suitably cleared subject before physically purging them from the database.)

3. Apply the following *?-replacement rule* to R_c:

   Let t be a tuple in R_c with $t[A_k]$ = "?." There are two cases:

   (a) There is a tuple $u \in R_c$ with $u[A_1, C_1] = t[A_1, C_1]$ and $TC[u] = t[C_k]$. In this case, we replace "?" in $t[A_k]$ by $u[A_k]$.

   (b) There does not exist a tuple $u \in R_c$ with $t[A_1, C_1] = u[A_1, C_1]$ and $TC[u] = t[C_k]$. In this case, we replace "?" by "Null" in $t[A_k]$.

4. Finally, make R_c subsumption free by removing all tuples s such that for some $t \in R_c$ and for all $i = 1, ..., n$ either (i) $t[A_i, s_i] = s[A_i, s_i]$ or (ii) $t[A_i] \neq$ Null and $s[A_i]$ = Null.

Examples. In this section, we give several examples to illustrate the update semantics as well as the decomposition and recovery algorithms.

The INSERT statement. To illustrate how the INSERT statement works, consider SOD_U and D_U as shown in Figure 28. Suppose a U-user wishes to insert a second tuple into SOD_U. He does so by executing the following INSERT statement:

```
INSERT
INTO      SOD
VALUES    ('Voy', 'Exp', 'Mars')
```

As a result of the above INSERT statement, SOD_U and D_U will change to the relations shown in Figure 29. If we wish to recover SOD_U, after step 1 of the recovery algorithm, SOD_U is identical to D_U of Figure 29. Since steps 2, 3, and 4 of the recovery algorithm make no changes to SOD_U, we have the desired result.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

Figure 28. SOD_U and D_U.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Voy	U	Exp	U	Mars	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U
Voy	U	Exp	U	Mars	U

Figure 29. SOD_U and D_U after INSERT.

The UPDATE statement. To illustrate the effect of an UPDATE statement, consider the instance SOD_U and the corresponding base relation D_U, given in Figure 30. Let the instance SOD_S be identical to SOD_U, in which case D_S is empty, as shown in Figure 31. Suppose an S-user makes the following update to SOD_S:

```
UPDATE    SOD
SET       DEST = 'Rigel'
WHERE     SHIP = 'Ent'
```

Using the update semantics, SOD_S will have one tuple, as shown in Figure 32, and by step 1 of the decomposition algorithm, D_S, which was empty prior to this update, will have a single tuple, call it *u*, as shown in Figure 32. Notice that *u* contains the pair <?, U>, which indicates that during the recovery "?" is to be replaced by the attribute value in the corresponding U-tuple. Specifically, let us use the recovery algorithm to reconstruct SOD_S. The first step of the algorithm forms the union of relations D_U and D_S in Figures 30 and 32. Since the key deletion rule does not apply, we move to step 3 (?-replacement rule) of the recovery algorithm, which will replace <?, U> in *u* by <Exp, U> (that is, the corresponding attribute values for 'Ent' in the lower level relation D_U in Figure 30). After the union is made subsumption free (step 4), we end up with the instance SOD_S in Figure 32, as desired.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Null	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Null	U

Figure 30. SOD_U and D_U.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Null	U	U

SHIP	OBJ	DEST

Figure 31. SOD_S and D_S.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	?	U	Rigel	S

Figure 32. SOD_S and D_S after UPDATE by S-user.

Next, suppose a U-user executes the following command against SOD_U, shown in Figure 30:

```
UPDATE  SOD
SET     DEST = 'Talos'
WHERE   SHIP = 'Ent'
```

As a result of this update, the decomposition algorithm only modifies D_U from the instance in Figure 30 to the one in Figure 33. Readers should verify that if we use the recovery to obtain SOD_S, we obtain the instance given in Figure 34, although no changes were made to the underlying D_S as a result of the above update. Of course, SOD_U will change to the relation shown in Figure 33.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U

SHIP		OBJ		DEST	
Ent	U	Exp	U	Talos	U

Figure 33. SOD_U and D_U after UPDATE by U-user.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	?	U	Rigel	S

Figure 34. SOD_S and D_S after UPDATE by U-user.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Spy	S	Rigel	S	S

SHIP		OBJ		DEST	
Ent	U	Spy	S	Rigel	S

Figure 35. SOD_S and D_S after UPDATE by S-user.

Finally, suppose starting with the instance SOD_S shown in Figure 34, an S-user invokes the following update:

```
UPDATE  SOD
SET     OBJ = 'Spy'
WHERE   SHIP = 'Ent' AND
        DEST = 'Rigel'
```

Using the update semantics, the SOD_S will change to the instance given in Figure 35, *not* to the instance given in Figure 36.

This follows from our underlying philosophy: We need to polyinstantiate either to close a signaling channel or to provide a cover story. In terms of

the decomposition, D_S will change from the instance in Figure 34 to the one in Figure 35. We leave it to the reader to verify that the recovery algorithm operates correctly.

SHIP		OBJ		DEST		TC
Ent	U	Exp	U	Talos	U	U
Ent	U	Exp	U	Rigel	S	S
Ent	U	Spy	S	Rigel	S	S

Figure 36. A multilevel relation different from the one in Figure 35.

The DELETE statement. To illustrate how the DELETE statement works, suppose a U-user executes the following DELETE statement against the relation SOD_U shown in Figure 33 (assume S-users see the instance given in Figure 34, D_U is as in Figure 33, and D_S is as in Figure 34):

```
DELETE
FROM    SOD
WHERE   SHIP = 'Ent'
```

Following the delete semantics, not only will SOD_U become empty, but SOD_S will become empty as well. As a consequence of the above DELETE, the decomposition algorithm will make D_U in Figure 33 empty. The reader should verify that although D_S (shown in Figure 34) does not change, if we were to recover SOD_S at this point, the key deletion rule in the recovery algorithm will delete the tuple for the starship 'Ent.'

Options and extensions. As we indicated earlier, the core integrity properties do not uniquely determine how an update by a c-user to R_c should be propagated to $R_{c'>c}$, and several different options have been proposed. This section discusses the relationship between the algorithms and these options.

The algorithms can accommodate the SeaView MVD requirement [DENN87, DENN88a, LUNT90] most easily. No changes are required in the decomposition algorithm; only the recovery algorithm needs to be modified. Steps 1 and 2 of the recovery algorithm remain the same as before. Steps 3 and 4 are changed as follows:

3′. For each i, $2 \le i \le n$, repeat the following:

Whenever t_1 and t_2 are two tuples in R_c such that $t_1[A_1, C_1] = t_2[A_1, C_1]$, add to R_c tuples t_3 and t_4, defined as follows:

$$
\begin{aligned}
t_3[A_1, C_1] &= t_1[A_1, C_1] \\
t_3[A_i, C_i] &= t_1[A_i, C_i] \\
t_3[A_j, C_j] &= t_2[A_j, C_j],\ 1 < j \leq n,\ j \neq i \\
t_4[A_1, C_1] &= t_1[A_1, C_1] \\
t_4[A_i, C_i] &= t_2[A_i, C_i] \\
t_4[A_j, C_j] &= t_1[A_j, C_j],\ 1 < j \leq n,\ j \neq i
\end{aligned}
$$

4′. Delete from R_c any tuple that has a "?" as a value.
5′. Step 5′ is the same as step 4 of the original algorithm.

The decomposition and recovery algorithms will have to be modified to accommodate the single tuple per tuple class approach [SAND90a] or the closely related single maintenance level attribute approach adopted by the LDV model [HAIG91a, STAC90]. These modifications are straightforward.

This brings us to the dynamic MVD requirement [LUNT91]. It too will require modifications to both the decomposition and recovery algorithms along the lines discussed elsewhere [JAJO91a]. The major difference is that in the single-level relations D_c, we will sometimes require "?" for classification attributes (rather than just for data attributes, as shown earlier).

It is also possible to have a single decomposition algorithm for updates and realize the several alternate semantics discussed above (and others from the literature) by varying only the recovery algorithm.

Conclusion

In this essay, we have examined the entity integrity requirement and the semantics of various update operations in the context of multilevel relations. These concepts were suitably generalized to deal with polyinstantiation. We have also described a decomposition algorithm that breaks a multilevel relation into single-level relations and a recovery algorithm that reconstructs the original multilevel relation from the decomposed single-level relations.

We believe much interesting work remains to be done in this area [JAJO90e]. In particular, we would like to give a complete and formal set of principles that can help with design and implementation of multilevel secure relational DBMSs. Initial steps have been taken in this direction in the present essay, but more remains to be done.

Acknowledgments

This work was partially supported by the US Air Force, Rome Air Development, through subcontract #C/UB-49; D.O. No. 0042 of prime contract #F-30602-88-D-0026, Task B-O-3610, with CALSPAN-UB Research Center. We are indebted to Joe Giordano and RADC for making this work possible.