IMPLEMENTING QUERIES AND UPDATES ON UNIVERSAL SCHEME INTERFACES

C. LECLUSE and N. SPYRATOS
Université de Paris Sud
91405 Orsay cedex
FRANCE

ABSTRACT: Using partition semantics [S84,LS87], we show that to every relational universe U and set of functional dependencies F, there corresponds a unique database scheme (called the canonical scheme) such that every query on the universe can be answered uniquely by a relational expression on the canonical scheme, and every update of the universal relation can be translated uniquely into a transaction on the canonical scheme. Our results render the relational model logically independent with respect to both queries *and updates* thus subsuming previous approaches to the problem [MRSSW87].

1. INTRODUCTION

As noted by Maier,Vardi and Ullman [MUV84], the relational data model has gone far toward *physical* data independence, but has not achieved the goal of *logical* data independence. That is, users of relational systems are relieved of specifying access paths within the structure of a single relation, but they still must navigate between relations. Users and application programs are protected from changes in the physical implementation of relations, but not from changes in the logical structure of a database, such as decomposition made for normalization or efficiency reasons.

Universal scheme interfaces are an attempt to achieve logical data independence. In a universal scheme interface, all the semantics of the database is loaded onto the attributes. Queries are phrased in terms of attributes alone and the user does not need to know which attributes are in which relations. That is, in a universal scheme interface, a database is presented as a semantic whole, accessible through its attributes alone.

The idea that data can, at least in principle, be thought of as residing in a single relation is an intuitively appealing one. In fact, even in the normalization process, the implicit assumption is that it makes sense to talk about attributes disembodied from any particular relation scheme and, therefore, with a meaning of their own. However, what this meaning should be was not clear, and this gave rise to much controversy. The breakthrough was Mendelzon's "weak" or "representative" instance view of universal relations [M84] exploited by Sagiv [S83] in a real system. As with dependencies, there are in fact many sound ways one can view the universal relation; Maier, Rozenshtein and Warren [MRW86] is a key paper integrating these differing viewpoints. They also survey a large number of systems that are, implicitely or explicitely, based on a universal scheme interface.

The weak instance view of universal relations seems to provide the right framework for querying universal scheme interfaces. However, there does not seem to be the flexibility to update over arbitrary

Proceedings of the 14th VLDB Conference
Los Angeles, California 1988

schemes that there is to query over arbitrary schemes. Proposals to remedy this defficiency (such as introducing "missing value" nulls) have led to increased complexity without really solving the update problem.

In this paper, we propose a novel approach for processing queries and updates in a universal scheme interface, based on the set-theoretic semantics of the partition model [S84,CKS86]. In our approach, the implementation of a universal scheme interface is done in two parts as follows:

DESIGN PHASE :
Given universe U and set of functional dependencies F, we first associate U and F with a (uniquely defined) database scheme, called the *canonical scheme* of U and F. Then, we associate every nonempty subset Q of U with three (uniquely defined) objects: a relational expression, an insertion transaction and a deletion transaction on the canonical scheme.

RUN TIME :
The data is stored A query or update, expressed in terms of a set of attributes Q, is processed as follows:

query: The relational expression associated, during the design phase, with Q is evaluated on the current database.

insertion: The insertion transaction associated, during the design phase, with Q is evoked and executed on the current database.

deletion: The deletion transaction associated, during the design phase, with Q is evoked and executed on the current database.

Therefore, our approach of implementing a universal scheme interface can be seen as adding an upper layer on a traditional DBMS. This upper layer is currently being developped in our laboratory.

The paper is organized as follows. In Section 2, we recall the basic definitions from the partition model [LS87] which provides the underlying semantics of our approach. In Section 3, we define partition semantics for queries *and updates* in a universal scheme interface; we show that, in terms of queries, our semantics is equivalent to weak instance semantics. In Section 4, we show how partition semantics can lead to unambiguous semantics for universal scheme interfaces for both queries and updates. Section 5 contains some conclusions and suggestions for further work.

2. PARTITION SEMANTICS.

2.1 Informal overview.

Consider the following database containing only two tuples:

AGE	SITUATION
Young	*Unempl.*

SITUATION	SEX
Unempl.	*Female*

The tuple *Young Unemployed* can be seen as a string of two uninterpreted symbols, *Young* and *Unemployed*. Now, think of a possible world, and let Ω be the set of all individuals in that world. Moreover, let I(*Young*) be the set of all individuals of Ω that are young, and call I(*Young*) the interpretation of *Young*. Similarly, let I(*Unemployed*) be the set of all individuals of Ω that are unemployed and call I(*Unemployed*) the interpretation of *Unemployed*. Clearly, the intersection I(*Young*)∩I(*Unemployed*) is the set of all individuals of Ω that are both young and unemployed. It is precisely this intersection that we define to be the interpretation of the tuple *Young Unemployed*. In other words, the interpretation of a tuple is the intersection of the interpretations of its constituent symbols.

This kind of set-theoretic semantics of tuples allows for a very intuitive notion of truth. A tuple t is called true in interpretation I iff I(t) is nonempty. Thus, for example, the (atomic) tuple *Young* is true iff I(*Young*) is nonempty, that is iff there is an individual in Ω which is young (at least one such individual). Similarly, the tuple *Unemployed* is true in I iff there is an individual in Ω which is unemployed. Finally, the tuple *Young Unemployed* is true in I iff there is an individual in Ω which is both young and unemployed.

The set-theoretic semantics just introduced allow for a very natural notion of inference through set-containment. To see this, consider the following question:

Assuming that *Young Unemployed* is true in I and that *Unemployed Female* is true in I, can we infer that *Young Unemployed Female* is true in I ?

If we recall the definition of truth given earlier, then we can reformulate this question as follows:

Assuming that I(*Young*) $\cap$ I(*Unemployed*) is not empty and that I(*Unemployed*) $\cap$ I(*Female*) is not empty, can we infer that I(*Young*) $\cap$ I(*Unemployed*) $\cap$ I(*Female*) is not empty ?

Clearly the answer depends on the interpretation I. However, if we impose constraints on the interpretation I, for instance, if we require that

$$I(Unemployed) \subseteq I(Young) \quad (1)$$

meaning that all unemployed individuals are young, or that

$$I(Unemployed) \subseteq I(Female) \quad (2)$$

meaning that all unemployed individuals are female, then the answer is yes.

As we shall see shortly, constraints such as (1) and (2) provide the right interpretation for functional dependencies. It is important to note that the tuple appearing in a database, such as *young Unemployed* and *Unemployed Female* are assumed to be true. Using constraints such as (1) and (2) above, we may discover that other tuples that do not appear in the database, such as *Young Unemployed Female* are also true. Given a database D, we shall be interested in the set T(D) of all tuples that are implied by D. We shall look at this set as the information content of D and we shall call two database equivalent if they have the same information content. What we shall be querying and updating is the information T(D) and not particular representation of this information.

2.2 The Model.

We shall consider, separately the syntax and the semantics of our model. The syntactic part is essentially the relational model. The semantic part is a formalization of the concepts explained above.

2.2.1 Syntax

We begin with a finite, nonempty set $U=\{A_1,\dots,A_n\}$. The set U is called the *universe* and the A_i's are called the *attributes*. Each attribute A_i is associated with a countably infinite set of symbols (or values) called the *domain* of A_i and denoted by dom(A_i). We assume that U $\cap$ dom(A_i) is empty for all i, and that dom(A_i) $\cap$ dom(A_j) = $\emptyset$ for $i \neq j$. A *relation scheme* over U is a nonempty subset of U; we call sch(U) the set of all realtion schemes over U and we denote a relation scheme by the juxtaposition of its attributes (in any order). A *tuple* t over a relation scheme R is a function defined on R such that $t(A_i)$ is in dom(A_i), for all A_i in R. We denote by dom(R) the set of all tuples over R. Clearly, dom(R) is the cartesian product of the domains of all the attributes in R. If t is a tuple over $R=A_1\dots A_n$, and if $t(A_j)=a_j$, $j=1,\dots,n$, then we denote the tuple t by $a_1\dots a_n$. A *relation* over R is a set of tuples over R.

Definition 2.1 A database over U is a pair (δ,F) such that

(1) δ is a function assigning to every relation scheme R over U a finite relation over R, and

(2) F is a set of ordered pairs (X,Y) such that X and Y are subsets of U.

Every pair (X,Y) is called a *functional dependency* and is denoted by $X \rightarrow Y$ ♦

Example 2.1 Consider a universe of three attributes, say U={A,B,C}, and let dom(A)=$\{a_1,a_2,\dots\}$, dom(B)=$\{b_1,b_2,\dots\}$, and dom(C)=$\{c_1,c_2,\dots\}$.

Define a function δ on relation schemes over U as follows:

$\delta(AB) = \{a_1b_1, a_2b_1\}$

$\delta(BC) = \{b_1c_1, b_1c_2\}$

$\delta(R) = \emptyset$ for all other schemes R.

Let F be the set $\{A \to B, BC \to A\}$.♦

I : $a_1 \to \{1,3\}$ $\quad a_2 \to \{2,4\}$

$b_1 \to \{1,2,3,4\}$ $\quad b_2 \to \{5,6\}$

$c_1 \to \{3\}$ $\quad c_2 \to \{1\}$

$x \to \emptyset$, for every x different than a_1,b_1,b_2,c_1,c_2.

$I(a_1b_1) = I(a_1) \cap I(b_1) = \{1,3\}$

$I(a_2b_1) = I(a_1) \cap I(b_1) = \{2,4\}$

$I(b_1c_1) = I(b_1) \cap I(c_1) = \{3\}$

$I(b_1c_2) = I(b_1) \cap I(c_2) = \{1\}$

$I(a_1b_1c_1) = I(a_1) \cap I(b_1) \cap I(c_1) = \{3\}$

$I(a_1b_1c_2) = I(a_1) \cap I(b_1) \cap I(c_2) = \{1\}$

FIGURE 2.1 An interpretation I of U

Given a database D=(δ,F), we call *scheme* of D, denoted by sch(D), the set of all relation schemes that are assigned non-empty relations under δ. That is, $sch(D) = \{R \in sch(U) \mid \delta(R) \neq \emptyset\}$. Thus, the database scheme in Example 2.1 consists of the relation schemes AB and BC.

2.2.2 Semantics.

We assume that the "real world" consists of a countably infinite set of objects, and we identify these objects with the positive integers. Let ω be the set of all positive integers, and let $2\omega = \{\tau / \tau \subseteq \omega\}$ be the set of all subsets of ω. The set 2ω is the semantic domain in which tuples and dependencies will receive their interpretations.

Throughout our discussions, we consider fixed the universe of attributes $U = \{A_1, A_2, \ldots A_n\}$, and the associated domains $dom(A_i)$. For notational convenience, we denote by SYMBOLS the union of all attribute domains and by TUPLES the union of all domains. That is:

$$SYMBOLS = \cup_{A \in U}\ dom(A),$$

$$TUPLES = \cup_{R \in sch(U)}\ dom(R).$$

Clearly, SYMBOLS is a subset of TUPLES.

Definition 2.2 An *interpretation* of U is a function I from SYMBOLS into 2ω such that

$$\forall A \in U, \forall a,a' \in dom(A), (a \neq a' \Rightarrow I(a) \cap I(a') = \emptyset)$$ ♦

Thus the basic property of an interpretation is that *different* symbols of the *same* domain are assigned disjoint sets of integers. In Figure 2.1, we see a function I satisfying this property. The intuitive motivation behind this definition is that an attribute value, say a, is an (atomic) property, and I(a) is a set of objects having property a under I. Furthermore, an object cannot have two different properties a, a' of the same "type"; hence $I(a) \cap I(a') = \emptyset$. Given an interpretation I, we can extend it from SYMBOLS to TUPLES as follows:

$$\forall R \in sch(U),\ \forall a_1a_2\ldots a_n \in dom(R),\ I(a_1a_2\ldots a_n) = I(a_1) \cap \ldots \cap I(a_n)$$

In Figure 2.1 above, we see some examples of computations of tuple interpretations. The intuitive motivation for this extension is that a tuple, say ab, is the conjunction of the (atomic) properties a and b. Accordingly, I(ab) is the set of objects having both properties a and b, and therefore, $I(ab) = I(a) \cap I(b)$. Our definition of an interpretation suggests intuitive notions of thruth and refinement for tuples, as follows:

Definition 2.3 Let I be an interpretation and let s,t be any tuples in TUPLES. We say that tuple t is *true* in I iff $I(t) \neq \emptyset$, and we say that tuple s *refines* tuple t in I iff $I(s) \subseteq I(t)$. ♦

That is, t is true in I if there is at least one individual having property t, and s refines t in I if every individual having property s (under I) also has property t. Note that every tuple t refines all its subtuples (for example, ab refines both a and b). We now define when an interpretation I is called a model of a database D.

Definition 2.4 Let $D = (\delta, F)$ be a database over U. An interpretation I of U is called a *model* of D if

(1) $\forall R \in sch(U),\ \forall t \in \delta(R),\ I(t) \neq \emptyset$.

(2) $\forall X \rightarrow Y \in F,\ \forall x \in dom(X),\ \forall y \in dom(Y),\ I(x) \cap I(y) \neq \emptyset \Rightarrow I(x) \subseteq I(y)$ ♦

Condition (1) of this definition says that every tuple appearing in the database must be true. Condition (2) says that every functional dependency must be interpreted as a function. Let us note that $X \rightarrow Y$ is still interpreted as a function even if we replace condition (2) by the stronger condition

(2') $\forall x \in dom(X),\ \exists y \in dom(Y),\ I(x) \subseteq I(y)$.

The difference between (2) and (2') is that condition (2) interprets $X \rightarrow Y$ as a partial function whereas condition (2') interprets $X \rightarrow Y$ as a total function. In this paper, we adopt the (more general) condition (2) in our definition of a model. We shall come back to this remark when discussing chase and weak chase in the following section.

Having defined the concept of model, we can now define the concept of consistency. A database D is called *consistent* iff D posseses at least one model; and otherwise D is called *inconsistent*. For example, the database $D = (\delta, F)$ of Example 2.1 is consistent as the interpretation shown in Figure 2.1 is a model of D. On the other hand, the following database is inconsistent, as no interpretation I can verify the tuples ab and ab', and the dependency $B \rightarrow A$, all at the same time.

AB	BC	
a b	b c	$B \rightarrow A$
a' b		$A \rightarrow C$

Definition 2.5 Let $D = (\delta, F)$ be a database over universe U. Let s and t be any tuples in TUPLES. We say that D *implies* t, denoted by $D \models t$, iff $m(t) \neq \emptyset$ for every model m of D. We say that D *implies* $s \leq t$, denoted by $D \models t \leq s$, if $m(t) \subseteq m(s)$, for every model m of D. ♦

Clearly, D implies all tuples appearing in D. This follows immediately from the definition of a model. On the other hand, the database D of Figure 2.1 does not imply the tuples $a_2b_1c_1$ and $a_2b_1c_2$, as they are false in the model I shown in the figure.

Given a model m of D, we denote by T(m) the set of all tuples in TUPLES which are true in m. Let us denote by mod(D) the set of all models of D, and let us define

$$T(D) = \cap_{m \in mod(D)} T(m).$$

T(D) is clearly the set of all tuples implied by D. In the following section, we use the set T(D) in order to define semantics for queries and updates in a universal scheme interface.

3. QUERIES AND UPDATES.

Our semantics for queries corresponds to the weak instance semantics, but our semantics for updates is new.

3.1 Queries.

We define queries as in a universal scheme interface. That is, a query is a set of attributes, plus a selection condition. However, in order to arrive at the formal definition of a query and its answer, we need some preliminary definitions and notations.

Given a universe U and a relation scheme Q over U, we call *elementary condition* over Q any expression of the form X=x, where X is a subset of Q and x a tuple over X.

Definition 3.1 We call *elementary query* over U, any expression of the form Q/X=x, such that Q is a relation scheme and X=x is an elementary condition over Q. Given a universe U, an elementary query Q/X=x, and a database D over U, the *answer* of Q/X=x with respect to D, denoted by $\alpha(Q/X=x,D)$, is defined as follows:

$$\alpha(Q/X=x,D) = \{t \in dom(Q) \,/\, D \models t,\ D \models t \leq x\}$$

♦

In other words, $\alpha(Q/X=x,D)$ is the set of all tuples over Q implied by D and refining x in D.

Example 3.1 Consider the universe U={A,B,C,D}, and a database D = ({ab,bc,cd}, {B→D, C→D}). Let m be any model of D. As cd is in D, we have $m(c) \subseteq m(d)$ because of the dependency C→D. Hence, we have $m(bcd) = m(b) \cap m(c) \cap m(d) = m(b) \cap m(c) = m(bc)$. As bc is in D, it follows that $m(bcd) \neq \emptyset$ and, thus, $m(bd) \neq \emptyset$. So D implies bd. Using ab, bd and the dependency B→D, we can show in the same way that D implies ad. So the answer to the query AD in the database D is {ad}. Notice that, without the dependencies B→D <u>and</u> C→D, the answer to the query AD would be empty! ♦

Clearly, in order to answer elementary queries, we must solve the following inference problems: Given a database D and tuples t and x,

(1) Decide whether $D \models t$,

(2) Decide whether $D \models t \leq x$.

A procedure for solving these problems is given in [LS87].

Elementary conditions can be combined using logical connectives in order to form more complex conditions. We call *selection condition* over Q, any combination of elementary conditions over Q, using logical connectives.

Definition 3.2 We call *query over U*, any expression of the form Q/s such that Q is a relation scheme and s is a selection condition. The answer to a query over U is defined recursively, based on the answers to elementary queries, as follows:

$$\alpha(Q/{-}{-}s,D) = \alpha(Q,D) -- \alpha(Q/s,D),$$
$$\alpha(Q/s \wedge s',D) = \alpha(Q/s,D) \cap \alpha(Q/s',D),$$
$$\alpha(Q/s \vee s',D) = \alpha(Q/s,D) \cup \alpha(Q/s',D).$$

♦

3.2 Updates.

We consider the problem of inserting and deleting a single tuple t in a database D=(δ,F). (In [LS87], we show that inserting or deleting a set of tuples is equivalent to inserting or deleting the tuples, one at a time). We assume that the set F is fixed, that is, we update δ and not F (ie, we update tuples and not dependencies). We denote by BASES(F), the set of all databases over a set F of functional dependencies. Given two databases D and D' in BASES(F), we say that D and D' are *equivalent*, denoted by D ≡ D', iff T(D) = T(D'). This definition is motivated by the fact that the information carried by a database D is the set T(D), namely the set of all tuples implied by D. We are interested mainly in the information T(D), and not in the representation of this information by different (but equivalent) databases. The set of all equivalence classes of BASES(F) is denoted by BASES(F)/≡ and the class of a database D is denoted by **D**. We shall take the set T(D) to be the representative of the class **D**. We say that a class **D** is *smaller* than a class **D'**, denoted by **D** < **D'**, iff $T(D) \subseteq T(D')$. What we are updating is the information T(D), rather than the database D. In other words, we are updating equivalence classes and *not* specific databases. As we update equivalence classes, we define insertion to be a function from the cartesian product BASES(F)/≡ x TUPLES into BASES(F)/≡. This function takes as arguments an equivalence class **D** and a tuple t, and returns an equivalence class **D'**. The conditions that the equivalence class D' must satisfy are stated formally in the following definition.

Definition 3.3 Let U be a universe and let F be a fixed set of functional dependencies over U. Define *insertion* of a tuple to be a function INS, from BASES(F)/≡ x TUPLES into BASES(F)/≡ , such that, for every tuple t and every database D:

(1) INS(**D**,t) is the smallest class **D'** verifying conditions (2) and (3) below:

(2) **D** < **D'**, and

(3) **D'** $\models$ t.

♦

Note that if **D'** exists then it is unique. Indeed, **D'** is the equivalence class that corresponds to

$$\cap \{T(D'') / T(D) \subseteq T(D''),\ t \in T(D'')\}$$

Thus INS is a well defined function. However, as insertion of tuples may create inconsistency with respect to F, it is clear that INS is only a partial function.

A word of explanation is in order here, concerning the requirement that INS(**D**,t) be minimal. First, observe that if INS(**D**,t) exists then there may be many different equivalence classes **D'** satisfying the conditions (2) and (3) above. Thus, in order for INS(**D**,t) to be a function, we must designate a unique class **D'** as the result of insertion. The reason why we ask for a minimal class **D'**, satisfying (2) and (3), is because we want to exclude undesirable insertions of tuples ("side effects"). For example, if D=({ab},∅) and we want to insert the tuple a'b', then any of the following non-equivalent databases satisfies conditions (2) and (3) above:

D' = ({ab,a'b'},∅), D'' = ({ab,a'b',a"b"},∅), ...

Of the above databases, condition (1) designates the minimal class **D'** as the result of the insertion. We define deletion of a tuple from an equivalence class in a similar manner.

Definition 3.4 Let U be a universe and let F be a set of functional dependencies over U. Define *deletion* of a tuple to be a function DEL from BASES(F)/≡ x TUPLES into BASES(F)/≡ , such that, for every class **D** and every tuple t,

(1) DEL(**D**,t) is the largest class **D'** verifying conditions (2) and (3) below:

(2) **D'** < **D**

(3) D' $\not\models$ s, <u>for all s such that: D $\models$ s≤t.</u>

♦

It is shown in [LS87] that DEL(**D**,t) is always defined, that is, DEL is a total function. The reason why DEL(**D**,t) is required to be a maxiamal class is similar (or, rather, symmetric) to the reason why INS(**D**,t) is required to be a minimal class. The following proposition describes a basic property of deletion, namely, when deleting a tuple t, we must also delete all its refinements (i.e., all its supertuples).

Proposition 3.1 If D is a database and t is any tuple, then we have:
$T(DEL(D,t)) = T(D) - \{s / s \in T(D),\ D \models s \leq t\}$.
Proof see [LS87]

Example 3.2 Let U = {A,B,C,D} be a universe, let F = {B→D, C→D} be a set of functional dependencies, and let D = {ab , bc}. We have T(D) = {ab, bc, a, b, c}.

<u>Insertion</u>: We consider the insertion of the tuple cd in D. A representative of the class INS(**D**,cd) is the database D' = {ab, bc, cd}, so our semantics for insertion corresponds to the intuitive notion of adding a tuple to a database. Moreover, we have T(D') = {abd, bcd, ab, bc, bd, cd, ad, a, b, c, d}.

<u>Deletion:</u> We consider the deletion of the tuple ad from the database D' above. D' $\models$ abd ≤ ad because abd is a super-tuple of ad. Moreover, D' $\models$ ab ≤ abd because of the dependency B→D. We deduce from

Proposition 3.1 above that the result of the deletion is the class D" such that T(D") = T(D')-{ab,abd, ad} = {bcd, bc, bd, cd, a, b, c, d}. A possible representative of this result is the databaseD" = {bc, cd, a}. ♦

It should be clear from definitions 3.3 and 3.4 that, in order to process insertion and deletion of tuples, all we need is a procedure for solving the following inference problems: Given a database D and tuples t and x,

(1) Decide whether $D \models t$,
(2) Decide whether $D \models t \leq x$

In other words, we have to solve precisely the *same* inference problems as in the case of queries! This is the reason why our decision procedure [LS87] strictly subsumes chase and weak chase, as we shall now explain.

3.3 Partition Semantics and Weak Instances.

Weak instances were first introduced as a means for discussing global satisfaction of a set of dependencies [M84] and has since been used in inferring missing information in a database state, discussing equivalence of database states, and defining Window Functions [MRW86]. Its semantics is captured by the well known chase procedure. The so-called weak-instance model seems to provide the right framework for querying universal scheme interfaces. However, there does not seem to be the flexibility to update over arbitrary schemes that there is to query over arbitrary schemes. In this paragraph, we compare the weak instance model, with our model. In the following, we denote by WChase a chase procedure in which a non distinguished variable can only be replaced by a distinguished variable and not by another non distinguished variable (this is what [MRW86] called 'null preserving chase'). We denote by $\rho(D)$ the tableau built from the database D by padding out the tuples of D with non distinguished variables. Finally, we denote by $\pi{\downarrow}_R(X)$ the set of all tuples in $\pi_R(X)$ containing no nulls (usually, $\pi{\downarrow}_R(X)$ is called *restricted projection* of X on R).

Theorem 3.1 Let $D=(\delta,F)$ be a database and let t be a tuple over a relation scheme R:

(1) D is consistent iff $\mathrm{WChase}_F(\rho(D))$ satisfies F.

(2) D implies t iff t is in $\pi{\downarrow}_R(\mathrm{Wchase}_F(\rho(D)))$. ♦

N.B. The terms "consistent" and "implies" have the sense of our model, described in Section 2 above, whereas the term "satisfies" has the sense of the relational model.

Proof see [LS87].

This theorem establishes the equivalence between our semantics and those of weak chase. However, our semantics is richer in two important ways. More precisely, functional dependencies can be interpreted in two ways in our model (as partial or total functions), and we can express both queries and updates over arbitrary schemes.

4. SYNTACTIC PROCESSING OF QUERIES AND UPDATES

In the previous section, we have seen that the same decision procedure (one that determines whether $D \models t$ and whether $D \models t \leq x$) is sufficient for processing both queries *and* updates, in a universal scheme interface. Thus, all we have to do is to implement such a procedure, in order to obtain a universal scheme interface. And, in fact, such a procedure is given in [LS87] and it has been implemented in C on a SUN worstation [M87]. However, this approach to implementing a universal scheme interface is, essentially, building a system from scratch.

In this paper, we propose a more pragmatic approach to the problem. Namely, rather than implementing a universal scheme interface from scratch, we propose to take advantage of existing

DBMS technology. The method that we propose for implementing a universal scheme interface can be decomposed into two parts, as follows:

DESIGN PHASE :
Given universe U and set of functional dependencies F, we first associate U and F with a (uniquely defined) database scheme, called the *canonical scheme* of U and F. Then, we associate every nonempty subset Q of U with three (uniquely defined) objects: a relational expression, an insertion transaction and a deletion transaction on the canonical scheme.

RUN TIME :
The data is stored according to the canonical scheme. User queries and updates are expressed in terms of attributes alone, and the user does not have to know which attributes are in which relations of the canonical scheme. A query or update, expressed in terms of a set of attributes Q, is processed as follows:

query: The relational expression associated, during the design phase, with Q is evaluated on the current database.

insertion: The insertion transaction associated, during the design phase, with Q is evoked and executed on the current database.

deletion: The deletion transaction associated, during the design phase, with Q is evoked and executed on the current database.

Therefore, our approach of implementing a universal scheme interface can be seen as adding an upper layer on a traditional DBMS. This upper layer is currently being developped in our laboratory.

4.1 Computing answers through relational expressions

Throughout our discussions, we consider fixed a universe U and a set of functional dependencies F. A *relational expression* over U is any well formed expression whose operators are relational algebra operators and whose operands are relation schemes over U. Given a database D and a relational expression e over a universe U, we can *evaluate* e over D by substituting database relations for the schemes in e and performing the operations. We denote the result of the evaluation by eval(e,D). Unfortunately, the evaluation of a relational expression does not always yield true tuples. To see this, consider the database D=({ab,bc},∅) and the expression e=AB BC. We have eval(e,D)={abc} although D does not imply abc! This example motivates the following definition:

Definition 4.1 Let F be a set of functional dependencies over a universe U. A relational expression e over U is called *sound* with respect to F iff every tuple in eval(e,D) is implied by D, for all D in BASES(F). ♦

As we have stated earlier, we are interested in answering queries by evaluating relational expressions. So, suppose that we want to answer a query Q/X=x over a database D in BASES(F). Moreover, suppose a family Δ of relation schemes over U whose union $\cup\Delta$ contains Q, and whose join Δ is a sound relational expression with respect to F. Then the expression $\sigma_{X=x}(\pi_Q(\Delta))$, when evaluated over D, produces tuples over Q that are implied by D and satisfy the condition X=x. It follows that these tuples are in the answer of Q/X=x. Motivated by this discussion, let us define a family Δ of relational schemes to be a *context* of Q iff $Q \subseteq \cup\Delta$ and $\pi_X(\Delta)$ is a sound relational expression. We denote by con(Q) the set of all contexts of Q. Let eval(Δ,D) denote the result of the evaluation of the expression $\sigma_{X=x}(\pi_Q(\Delta))$ over database D. It follows from our previous discussion that

$$\cup_{\Delta\in con(Q)} eval(\Delta,D) \subseteq \alpha(Q/X=x,D)$$

We shall show shortly (Theorem 4.5 below) that the inclusion also holds in the opposite direction, assuming that the database is complete. A database $D=(\delta,F)$ is called *complete* iff for all Q in sch(D), $\alpha(Q,D) = \delta(Q)$. That is, D is complete if for every Q in the schema of D, the answer to Q is precisely the relation stored under Q. So, in order to compute the answer $\alpha(Q/X=x,D)$, we must generate all contexts of Q during the design phase, and maintain a complete database during runtime.

We shall see how we maintain a complete database in Section 4.4 but first, let us see how we can generate contexts. The following theorem helps identifying contexts.

Theorem 4.1 Let R and S be relation schemes over U. If $R\cap S \rightarrow R$ is in F+, or $R\cap S \rightarrow S$ is in F+, then the expression R S is sound with respect to F. ♦

<u>N.B.</u> F+ denotes the closure of F under Armstrong axioms.

Proof see [LS88]

In view of this theorem, define relation schemes R and S to be *neighbors* (with respect to F) iff $R\cap S \rightarrow R$ is in F+, or $R\cap S \rightarrow S$ is in F+ . Consider, for example, the following universe U and set of functional dependencies F that we shall use as our running example:

$$U=\{A,B,C,D\} \quad , \quad F=\{B \rightarrow D, C \rightarrow D\}$$

Then, the relation schemes AB and BD are neighbors, as $B \rightarrow BD$ is in F+. Similarly, the relation schemes ABC and BCD are neighbors, as $BC \rightarrow BCD$ is in F+. Using the concept of neighbors, we can generate all contexts of a given relation scheme, as described in the following theorem.

Theorem 4.2 Let Q be a relation scheme, then

(1) {Q} is a context of Q.

(2) If Δ is a context of $Q\cup P$, then Δ is a context of Q

(3) If Δ is a context of Q, Δ' is a context of P, Q and P are neigbours,

then $\Delta \cup \Delta'$ is a context of $Q\cup P$.

(4) Nothing else is a context of Q.

Proof see [LS88]

In our running example, if we let Q=AD and apply Theorem 4.1, we find

con(AD) = { {AB,BD}, {AC,CD}, {ABC,BCD} , {AB, BC, CD}, {AD}, ... }

(Recall that con(Q) denotes the set of all contexts of Q).

4.2 The canonical scheme

We have seen so far how we could compute the answer of a query Q using the set of all contexts of Q. In this section, given a universe U and a set of functional dependencies F, we define a database scheme that we call canonical scheme and denote by can(U,F). This database scheme will be used to consider only "useful" contexts, as we shall see later.

In order to arrive at the definition of the canonical database scheme, we need some additional definitions and notations. First, define a family Δ of relational schemes to be *complete* iff $\forall X \in \Delta, \forall Y \subseteq X, Y \in \Delta$. For instance, in our running example, the family {AB,BC,A,B,C} is complete. Second, Define a context Δ of Q to be a *trivial context* of Q if there is P such that $Q \subseteq P$ and $P \in \Delta$. Now, define a family Γ to be *redundant* iff there is X in Γ, and non-trivial context Δ of X such that $\Delta \subseteq \Gamma$. For example, in our running example, the family
Γ = { ABD,AB,BD } is redundant, because {AB,BD} is a non-trivial context of ABD. Intuitively, as relations over ABD can be decomposed losslessly, over AB and BD, the scheme ABD is not needed. An example of

irredundant family is the following: {AB, BC, B}.

It is not difficult to see that the union of two complete and irredundant families is also complete and irredundant. It follows that the union of all complete and irredundant families is also complete and irredundant. In fact, this union is the greatest (with respect to set inclusion) complete and irredundant family over U and F. We call this family the *canonical scheme* over U and F and we denote it by can(U,F). For example, in our running example, the canonical scheme is {ABC, AB, AC, AD, BC, BD, CD, A, B, C, D}.

Theorem 4.3 Let U be a universe and F a set of functional dependencies. For every database D in BASES(F), there is a database D' over the canonical scheme which is equivalent to D. ♦

Proof see [LS88]

Finally, we need the notion of reduction of a relation scheme. Let X be a relation scheme. The *reduction* of X, denoted by red(X) is the family of all relation schemes Y such that:

(1) $Y \subseteq X$,

(2) $Y \rightarrow X$ is in F+.

(3) There is no $Y' \subseteq Y$ such that $Y' \rightarrow X$ is in F+.

For instance, in our running example, we have red(A) = {A}, red(BC) = {BC}, red(BD) = {B}, red(ABD) = {AB}.

The reductions of relation schemes have an interesting property, namely, they all belong to the canonical scheme, as stated in the following theorem.

Theorem 4.4 For every relation scheme X, we have

(1) red(X) $\subseteq$ can(U,F).

(2) $X \in$ can(U,F) $\Leftrightarrow$ $X \in$ red(X) ♦

Proof see [LS88]

We are now ready to define the implementation of queries, insertions, and deletions in a universal scheme interface.

4.3 Queries

Let U be a universe, let F be a set of functional dependencies, and let can(F,U) be the associated canonical scheme. Given a query q=(Q/X=x), we are looking for a relational expression e_q such that α(Q,D) = eval(e_q,D), for all D in BASES(F). We have the following theorem:

Theorem 4.5: Let q=(Q/X=x) be an elementary query over U.

$$\text{Let } e_q = \bigcup_{\substack{\Delta \in \text{con}(Q) \\ \Delta \subseteq \text{can}(U,F)}} \sigma_{X=x} (\pi_Q (\ \Delta))$$

Then for every complete database D over the canonical scheme, we have:

$$\alpha(q,D) = \text{eval}(e_q,D). \ \blacklozenge$$

Proof see [LS88]

For instance, in our running example, consider the query q=(BC/B=b). We compute first the set of all contexts of BC and the canonical scheme associated to U and F :

con(BC) = { {BC}, {BD, CD}, {ABC}, {BCD}}

can(U,F) = {ABC, AB, AC, AD, BC, BD, CD, A, B, C, D}.

It follows that the relational expression associated to q is

$$e_q = [\sigma_{B=b}\ \Pi_{BC}\ (BD \quad CD)] \cup [\sigma_{B=b}\ \Pi_{BC}\ (ABC)] \cup [\sigma_{B=b}(BC)].$$

Several systems have been proposed, based on universal scheme interfaces, such as SYSTEM/U [KFGU84], or PIQUE [MRSSW87], allowing the user to query a relational database, without any knowledge of its logical structure. The PIQUE language, in particular, provides a very convenient way for querying universal scheme interfaces. However, two major problems still remain with such systems. First of all, none of them allows the user to *update* over arbitrary schemes. Second, these systems lack a precise semantics for data, at the tuple level. As a consequence, the system must still make a choice between different possible access paths. In PIQUE, this problem is delegated to the so-called "Object Based Generator" which is in charge to provide a relational expression for representing a relation scheme. This Object Based Generator does not take semantic information into account, such as functional dependencies. In conclusion, existing systems do not go far enough toward logical data independence which is the final goal of universal scheme interfaces. We believe that only a semantic approach, such as ours, can reach this goal.

4.4 Updates

Let us call canonical transaction any mapping T that takes as arguments a complete database D over the canonical scheme and a tuple t, and returns a complete database T(D,t) over the canonical scheme. Such a mapping must be viewed as a sequence of queries, insertions and deletions cn relation schemes of the canonical scheme. It is important to note that the tuple t can be *any* tuple. That is, t does not have to be over a relation scheme of the canonical scheme.

Insertion

Let D be a database on the canonical scheme, and let t be a tuple to be inserted in D. We are looking for a canonical transaction T_{INS} such that: T_{INS} (D,t) is in the class INS(**D**,t). We have the following theorem:

Theorem 4.6 Let D=(δ,F) be a database over the canonical scheme. Let t be a tuple over relation scheme Q (NB. Q is not necessarily in the canonical scheme). If INS(**D**,t) is defined, then the following canonical transaction T_{INS}(D,t) leads to a complete database in INS(D,t).

```
For all X in can(U,F) such that $X \subseteq Q$
    insert $\Pi_X(t)$ in $\delta(X)$.
While $\delta$ is modified do
    For all X in can(U,F)
        $\delta(X) = \delta(X) \cup \alpha(X,D)$ ♦
```

For instance, in our running example, suppose that we want to insert the tuple abd. Moreover, suppose that the "current" database D is empty so that INS(**D**,abd) is defined. The corresponding canonical transaction is the following:

T_{INS}(D,abd) : insert the tuples a, b, ad, a, b and d in the corresponding relations.

Deletions

Let D be a database on the canonical scheme, and let t be a tuple to be deleted from D. We are looking for a canonical transaction T_{DEL} such that: T_{DEL} (D,t) is in the class DEL(**D**,t). We have the following theorem:

Theorem 4.7 Let D be a database over the canonical scheme. Let t be a tuple over relation scheme Q (NB. Q is not necessarily in the canonical scheme). DEL(**D**,t) is always defined and the following transaction T_{DEL} (D,t) leads to a complete database in DEL(**D**,t).

```
For all X such that $Q \subseteq X$,
  For all s in $\alpha(X/Q=t,D)$,
    For all Y in red(X),
      delete $\Pi_Y(s)$ from $\delta(Y)$, and
      For all Z such that $Z \subseteq Y$ and $Z \neq Y$
        insert $\Pi_Z(s)$ in $\delta(Z)$ ♦
```

Proof (sketch)
Using the semantics defined in Section 3, we must remove from the database all tuples t' such that D

implies $t' \leq t$. All tuples in $\alpha(X/Q=t,D)$,for a superset X of Q, satisfy this condition. We could show that if s is a tuple over X, and if Y is a relation scheme in red(X), then $m(s)=m(\Pi_Y(s))$ for all models m of D. So, looking at Theorem 4.3, it is necessary and sufficient to remove all tuples $\Pi_Y(s)$ from the corresponding relations in order to delete s. Now, every (strict) subtuple of $\Pi_Y(s)$ is inserted, in order to capture the semantics of Section 3 (the maximality condition). Notice that all subsets Z of Y are in can(U,F), because Y is in can(U,F), from Theorem 4.3.

For instance, in our running example, suppose that we want to delete ad, from the database D (containing tuples ab, bc, and cd). Following Theorem 4.6, the corresponding transaction is $T_{DEL}(D,ad)$:
for X=ADB, we have $\alpha(ADB/AD=ad,D) = \{adb\}$, and $red(ADB) = \{AB\}$, so we delete ab from $\delta(AB)$, and insert a and b in the corresponding relations;
for X=ADC, we have $\alpha(ADC/AD=ad,D) = \emptyset$;
for X=ADBC, we have $\alpha(ADBC/AD=ad,D) = \emptyset$.
We thus obtain a complete database containing the tuples a,b,bc, and cd. We can compare this result with the example 3.2 of Section 3.

5. CONCLUSIONS

We have used partition semantics in order to implement queries and updates on a universal scheme interface, through relational expressions and transactions on a traditional DBMS. Implementations of queries, based on the weak instance model, have been proposed in the past. However, proposals for the implementation of updates have only led to increased complexity without really soving the problem. We believe that it is in the implementation of updates that our method offers a simple solution.

Maier, Ullman and Vardi have schown [MUV84] that the effect of the representative instance, and therefore the partition semantics, cannot be simulated in the general case by first order expressions. In this paper, we have shown that it is possible for complete databases over canonical schemes. We have also shown that every database is equivalent with respect to queries to such a database. We have thus obtained a solution to the general case as follows: given U and F, we produce the canonical schema corresponding to U and F, and we simulate the effect of partition semantics by relational expressions over complete databases on that canonical schema.

The approach described in this paper is currently being implemented in the form of an upper layer for traditional DBMS's.

BIBLIOGRAPHY

[CKS86] Cosmadakis S., Kanellakis P.C., Spyratos N., Partition Semantics for Relations, ACM PODS, March 1985.

[KFGU84] Korth H.F., Kuper G., Feigenbaum J., VanGelger A., Ullman J.D., System U: A Database System Based on the Universal Relation Assumption. ACM TODS, 9:3, September 1984.

[LS87] Lécluse C., Spyratos N. Updating Weak Instances Using Partition Semantics, Rapport de Recherche LRI, No 364, July 1987.

[LS88] Lécluse C., Spyratos N. Implementing Queries and Updates in Universal Schemes Interfaces, INRIA Resaerch Report No 805, March 88

[M84] Mendelzon, AO. Database states and their tableaux, ACM TODS 9:2 p264-282, 1984.

[M87] Mancuso F., Implémentation d'un modèle de données ensembliste. Rapport de stage de DEA, Université de Paris Sud, Orsay, Septembre 1987.

[MRW86] Maier D., Rozenshtein D., Warren DS., Window functions, in Advances in

Computer Research 3, p213-246, JAI Press, London.

[MRSSW87] Maier D., Rozenshtein D., Salveter S., Stein J., Warren D., Pique: A Relationnal Query Language Without Relations, in Information Systems, Vol 12, No 3, 1987.

[MUV84] Maier D., Ullman J.D., Vardi M.Y., On the Foundations of the Universal Relation Model, ACM TODS, Vol 9:2, 1984.

[S84] Spyratos N., The partition model - a deductive database model. INRIA Research Report 1984, ACM TODS, March 1987.

[S83] Sagiv Y., A Characterisation of Globally Consistent Databases and their Correct Acess Path. ACM TODS 8:2, p266-286, 1983.