Identifying and Update of Derived Functions in Functional Databases

Ramana Yerneni   Sitaram Lanka
Department of Computer Science
Penn State University
University Park, PA 16802
U.S.A

Abstract

The aim of this paper is to investigate the consistency problems that arise in the design and implementation of functional databases [1] [2] [3]. Redundancy is a common feature in the specification of functional databases [3]. The notion of derived functions is used to capture these redundancies. Maintaining the consistency of databases in the presence of redundant specification is a difficult task. To perform this task the derived functions in the conceptual schema have to be identified, and updates on derived functions should be performed in a manner consistent with their derivations. An interactive design aid is developed to facilitate the identification of derived functions. We have also developed algorithms that perform updates in a side effect free manner.

1 Introduction

Functional data models have been reported in [1] [2] [3]. A functional database is a database based on the functional data model. We can envision a functional database as a set of object types (entity types) together with a set of functions that operate on these object types. The functions can be expressed as,

$$F : \alpha \rightarrow \beta \tag{1}$$

where F is the name of the function whose domain consists of objects of type α and whose range consists of objects of type β. Note that (1) refers to a function definition. Typically, whenever we are referring to a function in the context of the conceptual schema we mean its function definition. But when operations on functions are being referred to, we mean the actual functions (sets of pairs of objects). Also, note that functions as in (1) are not necessarily single-valued functions. In this sense, these functions are mappings or relations. A conceptual schema of a functional database is a collection of functions[1]. For example, consider the conceptual schema S1 shown in table 1. The con-

1. grade: [student; course] → letter_grade
2. score: [student; course] → marks
3. cutoff: marks → letter_grade
4. teach: faculty → course
5. taught_by: course → faculty

Table 1: Conceptual Schema S1

ceptual schema consists of syntactic information of functions in the form of their definitions. The underlying semantics of these functions are not specified directly in the conceptual schema. The functions in a conceptual schema are of two types – base and derived. Base functions are usually extensionally stored (i.e., stored internally as a table) and derived functions intensionally stored (i.e., computed using an algorithm). The derived functions can be obtained from the base functions by applying certain operations. A derivation of a derived function is an ordered sequence of base functions along with the appropriate operations, which specifies a method of obtaining the derived function from these base functions. Composition and inverse are the two most important operations in such derivations. Composition is denoted by '∘ ' and is defined as x:(f ∘ g) = (x:f):g. Inverses of functions are defined as: f is the inverse of g iff f = {<a,b>| <b,a>∈ g}.

The distinction between base and derived functions follows from the assumed underlying semantics. For example, under the assumed semantics, *grade* may be derived from the composition of *score* and *cutoff* (grade = score ∘ cutoff). Two related problems in the presence of derived functions in functional databases are: (i) separating base and derived functions, and identifying the derivations of derived functions, (ii) updating the derived functions in a consistent manner. In this paper we will address these two problems.

Currently, in the functional data model [3] it is assumed that the derived functions along with their derivations are explicitly specified in the conceptual schema by the designer. However, while designing a nontrivial database, it is difficult to ensure that all the underlying semantics are properly reflected in such a conceptual schema specification. In the absence of proper verification this may lead to an inconsistent database. For example, one can define a conceptual schema with *grade, score* and *cutoff*, none of them marked as derived. If the underlying semantics imply that *grade* = *score* ∘ *cutoff*, updating *grade* alone without reflecting appropriate changes in *score* and *cutoff* will render the database inconsistent. This forms the motivation for developing a design aid that assists in the identification and verification of derived functions and their derivations.

Presently, in functional databases updates on derived functions are disallowed [3]. Derived functions are defined only for querying facility. The philosophy of functional databases is to provide a high level abstraction of the information content in the form of functions. Relegating derived functions to a second class status is against this basic philosophy.

Updating derived functions is similar to updating views in a relational database [4] [5] [6]. Updating through views gives rise to ambiguous information. Current solutions [6] [7] [8] [9] do not handle such ambiguous information in a satisfactory

[1]Strictly speaking, we should also include the various object types under consideration as part of the conceptual schema.

manner. They approximate the ambiguous information in the conventional database framework. In such a framework the only operations allowed on the data are the addition and removal of tuples. An update on a view is translated into a sequence of addition and removal of tuples in base relations which reflects the desired effect of the update. The "goodness" of the approximation is measured by quantifying the undesirable side effect. However, the addition and removal of base relation tuples do not follow from the definition of the view and the update of the view.

In this paper we will describe the kinds of ambiguous information generated by updates on derived functions and discuss how such information can be handled in an elegant manner. We develop update semantics which follow directly from the definition of derived functions. An algorithmic solution to the problem of updating functional databases in a consistent manner is advanced. Our algorithms are faithful to the proposed update semantics.

In section 2 we introduce the notion of a minimal schema, which facilitates the task of separating base and derived functions. A restricted class of conceptual schema for which this task can be performed in polynomial time is examined. In the general case, we propose an on-line design aid to identify the derived functions and their derivations. The problem of updating derived functions in a consistent manner is addressed in section 3. We will describe the update semantics for functional databases. In section 4 we advance algorithms to perform updates according to these semantics.

2 Minimal Schema Problem

The schema of a functional database (FDB) consists of the definitions of the functions of the FDB. It is useful to view the schema as a set of triplets, <function_name, domain_type, range_type> and the functions at an instance of the FDB as sets of pairs, <domain_val, range_val>[2]. We use $F_1, F_2, \ldots, F_n$ to denote the function definitions in a schema, and $f_1, f_2, \ldots, f_n$ to stand for their values at an instance. If I is an instance of an FDB $\mathcal{F}$ and $\mathcal{B}$ is any subschema of $\mathcal{F}$ then $\mathcal{B}$(I) denotes the set of functions of $\mathcal{B}$ at I. A function G is **derived** from $\{F_{i_1}, \ldots, F_{i_k}\}$ iff $\exists$ $u_1, \ldots, u_k \in$ {identity, inverse} such that $\forall$I, $g = u_1 f_{i_1} \circ \ldots \circ u_k f_{i_k}$. A **minimal schema** of an FDB with schema $\mathcal{S}$ is the minimal subschema $\mathcal{M}$ such that $\forall$G in $\mathcal{S}$, either G is in $\mathcal{M}$ or, $\exists$ $F_{i_1}, \ldots, F_{i_k}$ in $\mathcal{M}$ such that G is derived from $\{F_{i_1}, \ldots, F_{i_k}\}$. **The minimal schema problem** (MSP) is stated as: given an FDB $\mathcal{F}$, find a minimal schema of $\mathcal{F}$. Observe that a solution to the MSP will achieve the separation of base and derived functions in a conceptual schema. The minimal schema is the set of base functions of the FDB (the rest are derived).

2.1 Unique Form Assumption

The type functionality of a function indicates the nature of mapping it defines: one-one, one-many, many-one, and many-many. For example, the type functionality of *cutoff* is many-one meaning many marks can correspond to a letter_grade. We say two functions are syntactically equivalent if they have the same domain type and the same range type. Two functions are semantically equivalent if at every instance they have the same set of pairs of domain and range values. A sequence of functions along with appropriate operations is a derivation of a derived function if it is semantically equivalent to the derived function. For functions to be semantically equivalent it is necessary for them to be syntactically and type functionally equivalent. It is easy to verify syntactic and type functional equivalence based on information provided in the conceptual schema. However, it is not possible to test for complete semantic equivalence. This is because the underlying semantics of functions are not expressed in the conceptual schema. Thus the derived functions cannot be identified on the basis of purely syntactic information expressed in the conceptual schema. One way of overcoming this hurdle is to impose certain constraints on the specification of the conceptual schema which enable us to deduce the relevant semantic information from the available syntactic information.

Such a constraint is the *Unique form assumption.* We define the **closure** of a set of functions $\mathcal{G}$ as

$$< \mathcal{G} >= \{g \mid g = u_1(f_{i_1}) \circ u_2(f_{i_2}) \ldots \circ u_k(f_{i_k})\}$$

where $f_{i_j} \in \mathcal{G}$, $1 \le j \le k$, $u_i \in$ {identity, inverse}. $<\mathcal{G}>$ is a set of derived functions formed from $\mathcal{G}$ using composition and inverse. An instance I of an FDB with schema $\mathcal{S}$ is in **unique form** if and only if $((\forall\, h \in \mathcal{S}(I))(\forall f) \in <\mathcal{S}(I)>)$ [f and h are syntactically and type functionally equivalent implies f and h are semantically equivalent at I]. The UFA for an FDB $\mathcal{F}$ states that all possible instances of $\mathcal{F}$ are in unique form. The UFA provides the necessary semantic information (not directly specified in the conceptual schema) for separating the base and derived functions. In the absence of the UFA, it is not possible to decide whether a function is base or derived purely from the specification of the conceptual schema.

Lemma 1: In the absence of the UFA for an FDB $\mathcal{F}$ with schema $\mathcal{S}$, the minimal schema of $\mathcal{F}$ is $\mathcal{S}$.

Proof: Assume otherwise. Let $\mathcal{M}$ be any minimal schema of $\mathcal{F}$ such that $\mathcal{M} \neq \mathcal{S}$. Then there is a function f which is in $\mathcal{S}$ and is not in $\mathcal{M}$. Consider an instance I of $\mathcal{F}$ with f nonempty and every other function (of $\mathcal{S}$) empty. I is a possible instance of $\mathcal{F}$ in the absence of UFA for $\mathcal{F}$. Obviously, at I, f $\notin$ $<\mathcal{M}(I)>$ and hence $\mathcal{M}$ is not a minimal schema of $\mathcal{F}$. □

In what follows we describe how the UFA yields a polynomial algorithm for MSP. We define the **function graph** of an FDB $\mathcal{F}$ with schema $\mathcal{S}$ as an undirected graph $G_{\mathcal{F}}$= (V, E) where V is the set of object types of $\mathcal{F}$ (i.e., domains and ranges of the various functions) and E = {(D_1, D_2)| for some F in $\mathcal{S}$, F: $D_1 \rightarrow D_2$}. The syntax and type functionality of an edge follow from the function it represents. We define the syntax of a path $D_{i_1}, \ldots, D_{i_k}$ as $D_{i_1} \rightarrow D_{i_k}$. The type functionality of a path is the composition of the type functionality of the edges in the path.

Algorithm AMS:

```
Input: Schema $\mathcal{S}$of an FDB $\mathcal{F}$.
Output: Minimal schema $\mathcal{M}$ of $\mathcal{F}$.
Step 1: Construct $G_{\mathcal{F}}$, the function graph of $\mathcal{F}$.
Step 2: $\overline{M} = \phi$
    For each edge e $\in$ E do
        { if ($\exists$ a path p in $G' = (V, E - \overline{M} - \{e\})$ such that
        p is syntactically and type functionally equivalent to e)
```

[2]The functional operators – identity, composition and inverse – are applied on functions at instances of the FDB.

```
        then add e to $\overline{M}$ }
Setp 3: $\mathcal{M} = \mathcal{S}$ - $\overline{M}$
```

Lemma 2: AMS correctly computes a minimal schema.

Proof: Let $\mathcal{M}$ be the subschema outputted in step 3. Let H $\in$ $\mathcal{S}$ - $\mathcal{M}$. Since H $\notin$ $\mathcal{M}$, $\exists$ a path $D_{i_1}, \ldots D_{i_k}$ in $G_{\mathcal{M}}$ which is syntactically and type functionally equivalent to H. This means $\exists$ a set of functions $F_{i_1}, \ldots, F_{i_{k-1}} \in \mathcal{M}$ and $u_1, \ldots, u_{k-1}$ such that H is syntactically and type functionally equivalent to $u_1 F_{i_1} \circ \ldots \circ u_{k-1} F_{i_{k-1}}$. Under UFA this implies $\exists$ $F_{i_1}, \ldots, F_{i_{k-1}} \in \mathcal{M}$ such that H is derived from $\{F_{i_1}, \ldots, F_{i_{k-1}}\}$. Thus $\forall$ H $\in \mathcal{S}$, either H is in $\mathcal{M}$ or, $\exists$ $F_{i_1}, \ldots, F_{i_k}$ in $\mathcal{M}$ such that H is derived from $\{F_{i_1}, \ldots, F_{i_k}\}$.

Now consider any $\mathcal{M}' \subset \mathcal{M}$. Let H $\in \mathcal{M}$ - $\mathcal{M}'$. Since H $\in \mathcal{M}$, $\nexists$ a path in $G_{\mathcal{M}}$ (and so in $G_{\mathcal{M}'}$) which is syntactically and type functionally equivalent to H. An instance of $\mathcal{F}$ in which every function in $\mathcal{S}$ - {H} is empty and H is nonempty is valid under the UFA. Therefore, $\exists$ I such that H(I) $\neq u_1 f_{i_1}, \ldots u_k f_{i_k}$, for any $u_1, \ldots, u_k$ and $F_{i_1}, \ldots, F_{i_k}$, where $F_{i_j} \neq$ H. This implies that $\mathcal{M}'$ is not a minimal schema. Hence the lemma. □

Lemma 3: AMS takes $\mathcal{O}(n^2)$ time, where n is the number of functions.

Proof: Step 1 and 3 are computed in $\mathcal{O}(n)$ time. Step 2 consists of n iterations. Each iteration involves a search traversal of the function graph which takes $\mathcal{O}(n)$ time. □

We summarize these results in the following theorem.

Theorem 1: The minimal schema problem is solvable in polynomial time under UFA. □

The set of derivations of a derived function is given by the set of syntactic and type functionally equivalent paths in the function graph of the minimal schema. Thus under the UFA we can separate the base and derived functions and identify the derivations.

However, many naturally occuring conceptual schema cannot be admitted under the UFA. For example, consider the following conceptual schema S2.

$$\text{teach}: \text{faculty} \rightarrow \text{course}; \ (\text{many} - \text{many})$$
$$\text{class\_list}: \text{course} \rightarrow \text{student}; \ (\text{many} - \text{many})$$
$$\text{lecturer\_of}: \text{student} \rightarrow \text{faculty}; \ (\text{many} - \text{many})$$

The function *teach* corresponds to the list of courses taught by each faculty member, *class_list* gives the list of students enrolled in each course and, the list of faculty who lecture to a given student can be obtained from *lecturer_of*. One may obtain the set of lecturers of a student by finding his or her courses and obtaining the faculty teaching them, that is, *lecturer_of* and ($class\_list^{-1} \circ teach^{-1}$) are semantically equivalent. But given a course, all students whose lectures teach this course do not necessarily constitute the class_list of this course. In this sense ($teach^{-1} \circ lecturer\_of^{-1}$) and *class_list* are semantically different. Also, *teach* and ($lecturer\_of^{-1} \circ class\_list^{-1}$) are semantically different. Thus in S2, the only derived function is *lecturer_of*, and *teach* and *class_list* are its base functions. Under the UFA any of the three functions should be construed as derived because each of them are syntactically and type functionally equivalent to the composition of the other two. Hence such a conceptual schema under the assumed semantics is not allowed.

Purely based on systactic and type functional information in the conceptual schema, it is not possible to deduce the necessary semantic information which is required in identifying the derived functions. Such information can only be provided by the designer. In the following subsection we will describe an interactive design approach that elicits information from the designer to identify the derived functions and their derivations.

2.2 On-line Design Methodology

At the heart of the on-line design methodology a function graph is maintained dynamically. Initially we start with an empty graph and add the functions of the conceptual schema one at a time. At any given time during this process the function graph corresponds to the minimal schema of the set of functions added so far. This is achieved by manipulating the function graph appropriately when a new function is added. Such a manipulation of the function graph is described in the method below.

Method 2.1:

```
Goal: Dynamically maintain the minimal schema.
Step 1: Add the next function to the function graph.
Step 2: Identify all cycles formed by this function.
Step 3: For each cycle identified do
    (i) identify the candidate derived functions in the cycle.
    (ii) report these (cycle and candidate derived functions)
          to the designer.
    (iii) remove the edge specified by the designer.
Step 4: If more functions to be added then go to step 1.
```

Step 3 describes the corrective action taken on cycles in the conceptual schema by appropriate designer intervention. Since redundancies in the conceptual schema are characterised by cycles in the function graph, it is desirable to eliminate such cycles. A cycle can be broken if one of its edges is a derived function. A necessary condition for an edge to be a derived function is that its syntactic and type functional information agree with the other path between that pair of nodes in the cycle. Such an edge is termed a candidate derived function. The candidate derived functions of a cycle are found by simply traversing the cycle. The system provides the designer with the cycle and the candidate derived functions. The designer will decide on how to break the cycle.

If the function graph is maintained as an acyclic graph, then addition of a new function will result in atmost one cycle. This cycle can be found in $\mathcal{O}(n)$ time. The candidate derived functions of a cycle can be identified in $\mathcal{O}(l^2)$ time where l is the length of the cycle. Thus method 4.1 takes $\mathcal{O}(n^3)$ time (besides the dialogue with the designer). In the case of the function graph being cyclic, addition of an edge may result in an exponential number of cycles. In such a case the method is exponential.

Along with the dynamic function graph the system maintains a data structure that keeps track of the functions in the existing conceptual schema. Any function in this data structure which is not in the function graph is construed as a derived function; all other functions are base. Information regarding minimal schema, derived functions and their derivations can be extracted from the dynamic function graph and this data structure, at any juncture by the designer (typically at the end of the design).

If the function graph is maintained as an acyclic graph, each derived function has a unique derivation which is represented by the unique path between the respective pair of nodes. In the case of cyclic function graphs there can be multiple derivations for a derived function. To obtain the derivations of a derived function the system will first find all paths between its pair of nodes. Some of these paths may not correspond to actual derivations. Through designer intervention all such paths are filtered out and the correct derivations obtained.

2.3 An Example

Consider a trace of method 2.1 on the following conceptual schema. Assume the functions are added one at a time in the order shown below. Addition of a function may involve a trivial action of simply including it in the dynamic function graph as base function, or a nontrivial action of taking appropriate corrective measures on the cycles found. Only the nontrivial actions taken by the system are indicated below.

$$\text{teach}: \text{ faculty} \rightarrow \text{ course; (many} - \text{many)}$$

$$\text{taught\_by}: \text{ course} \rightarrow \text{ faculty; (many} - \text{many)}$$

The system finds a cycle with both *teach* and *taught_by* as candidate derived functions. The designer may specify *taught_by* to be removed (i.e., considered as derived function).

$$\text{class\_list}: \text{ course} \rightarrow \text{ student; (many} - \text{many)}$$

$$\text{lecturer\_of}: \text{ student} \rightarrow \text{ faculty; (many} - \text{many)}$$

The system finds the cycle, *teach - class_list - lecturer_of*, and identifies all three as candidate derived functions. The designer chooses the only correct derived function, *lecturer_of*, and instructs its removal.

$$\text{grade}: \text{ [student; course]} \rightarrow \text{ letter\_grade; (many} - \text{one)}$$
$$\text{attendance}: \text{ [student; course]} \rightarrow \text{ attn\_percentage; (many} - \text{one)}$$
$$\text{attendance\_eval}: \text{ attn\_percentage} \rightarrow \text{ letter\_grade; (many} - \text{one)}$$

The cycle formed by *grade, attendance, attendance_eval* is found and *grade* is reported as candidate derived function. The designer does not agree with the system and no edge is removed.

$$\text{score}: \text{ [student; course]} \rightarrow \text{ marks; (many} - \text{one)}$$
$$\text{cutoff}: \text{ marks} \rightarrow \text{ letter\_grade; (many} - \text{one)}$$

Now two cycles: *grade - score - cutoff* and *score - cutoff - attendance_eval - attendance* are identified. For the first cycle *grade* is identified as a candidate derived function and designer confirms its removal. For the second cycle no candidate derived function is reported.

At this juncture the function graph is shown in figure 1. The base functions are *teach, class_list, score, cutoff, attendance,* and *attendance_eval*; the derived functions are *taught_by, lecturer_of, grade.* The system, on the request of the designer, will report the following potential derivations.

taught_by = $teach^{-1}$; (confirmed by the designer).

lecturer_of = $class\_list^{-1} \circ teach^{-1}$; (confirmed by the designer).

grade = score $\circ$ cutoff; (confirmed by designer).

grade = attendance $\circ$ attendance_eval; (invalidated by the designer).

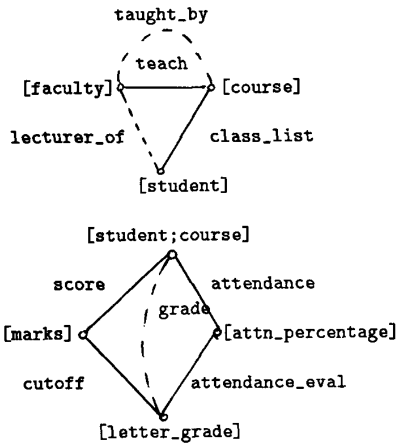

Figure 1: Dynamic Function Graph

3 Updating Functional Databases

Update in a functional database is performed on the functions of the conceptual schema. For sake of simplicity we consider updates a tuple at a time. A general update request can be viewed as a sequence of such simple updates. Update can be of three types: insert, delete, and replace and they are represented as: INS(f, $< x, y >$), DEL(f, $< x, y >$), and REP(f, $< x_1, y_1 >$, $< x_2, y_2 >$) respectively.

An update on a base function is directly effected on the extensionally stored table. An update on a derived function is translated into a corresponding sequence of updates on the base functions of its derivation. It is nontrivial to find a translation of a derived function update which achieves the desired effect, and at the same time, does not cause undesirable side effects. For example, consider the following functional database with conceptual schema – teach: faculty $\rightarrow$ course, class_list: course $\rightarrow$ student, and pupil: faculty $\rightarrow$ student. Pupil is a derived function with derivation teach $\circ$ class_list. An instance of $\mathcal{F}$ is {teach = { < euclid, math> , < laplace, math> < laplace physics> }, class_list = {< math, john> , < math, bill> }, pupil = { < euclid, john> , < euclid, bill> , < laplace, john> , < laplace, bill> }}. The following base updates, u_1: INS(class_list, < physics, bill>), and u_2: DEL(teach, < laplace, physics>) are handled by adding < physics, bill> to the stored table *class_list* and deleting < laplace, physics> from the stored table *teach*. Now consider u_3: DEL(pupil, < euclid, john>). One may attempt to achieve the desired effect by performing either DEL(teach, < euclid, math>) or DEL(class_list, < math, john>). However, observe that both of these have the undesirable side effect of deleting, from pupil, < euclid, bill> and

< laplace, john> , respectively.

3.1 Derived Update Problem

The problem of finding a translation for a given derived update which will achieve the desired effect with no side effects is termed "the derived update problem". The derived update problem is similar to the view update problem in relational databases as addressed in [8] [9] [7] [6] [10] [5] [11]. In [11] they have proposed a solution to the view update problem which depends on the semantic information elicited from the designer at view definition time as well as the user at update time. However, it is not clear what kind of information is relevant in the case of views involving the join operator. Since the most important operator in our derivations is composition (analog of join), [11] does not provide a solution to the derived update problem. In [6], a "correctness" criterion for view updates is formulated. According to this criterion an update on a view is "correctly" performed by a translation if the translation has the desired effect on the view and no side effect on it. A translation is said to have no side effect on the view if the symmetric difference of the extensions of the view before and after the update is equal to the set of tuples specified in the view update. [9] consider a database as a consistent theory, i.e., a collection of facts. Updates are carried out such that the new database differs minimally (in terms of number of facts deleted and number of facts inserted) from the old database. We illustrate the update semantics of [6] and [9] by the following example. Consider the conceptual schema: $r_1(AB)$, $r_2(BC)$, $r_3(CD)$, and the view $v_1(AD) = \pi_{AD}(r_1 \bowtie r_2 \bowtie r_3)$ with the following instance $r_1 = \{< a_1, b_1 > , < a_1, b_2 > \}$, $r_2 = \{< b_1, c_1 > , < b_2, c_1 > \}$, $r_3 = \{< c_1, d_1 > \}$, and $v_1 = \{< a_1, d_1 > \}$. Consider the update u_4: DEL(v_1, $< a_1, d_1 >$). A "correct" translation of this update under [6] semantics is DEL(r_1, $< a_1, b_1 >$), and DEL(r_1, $< a_1, b_2 >$). Such a sequence will have no side effect on the view v_1. According to the semantics of [9] u_4 is performed by deleting DEL(r_3, $< c_1, d_1 >$), because this is the only way which results in a new database that differs by exactly one fact (other than the fact to be deleted by u_4).

Note that the only information specified by the update is that $< a_1, d_1>$ does not belong to v_1. This does not imply the falsity of any base fact[3]. In this sense removing $< a_1, b_1>$ and $< a_1, b_2>$ from r_1 or $< c_1, d_1>$ from r_3 is unjustified. In the framework of [6] and [9], the only way an update is carried out is by adding and removing certain tuples from the database. However, in some situations it is not possible to capture the information specified through an update in this manner. For instance in the above example, the information that can be drawn from u_4 is $\neg(< a_1, b_1 > \in r_1 \wedge < b_1, c_1 > \in r_2 \wedge < c_1, d_1 > \in r_3)$[4]. Note that it is not possible to infer that any subset of the conjuncts are false. Such information is not captured in the update semantics of [6] and [9]. The same objection holds in the case of [8] and [7].

It is crucial for a database supporting view updates to handle information of the kind explained above. We propose new update semantics which capture such information and we will implement the update operation in a manner faithful to these semantics, in the framework of functional databases.

3.2 Update Semantics in Functional Databases

We percieve a functional database as a consistent set of facts in which logical implications specified through data dependencies (like functional dependencies), derivations of derived functions, etc., hold. Each fact is denoted as a triple $< f, a, b >$ which represents f(a) = b. Consider a functional database $\mathcal{F}$ with the functions F_1: A → B, F_2: B → C, and F_3: A → C; f_3 is a derived function with derivation $f_1 \circ f_2$. Note that the derivation is equivalent to the logical implications:

1. $< f_1, a, b > \wedge < f_2, b, c > \rightarrow < f_3, a, c >$
2. $< f_3, a, c > \rightarrow \exists$ b such that $< f_1, a, b > \wedge < f_2, b, c >$.

Let an instance of $\mathcal{F}$ be: $\{< f_1\ a_1\ b_1 > , < f_1\ a_2\ b_1 > , < f_2\ b_1\ c_1 > , < f_2\ b_1\ c_2 > , < f_3\ a_1\ c_1 > , < f_3\ a_1\ c_2 > , < f_3\ a_2\ c_1 > , < f_3\ a_2\ c_2> \}$. Consider an update u_5: DEL(f_3, $< a_1, c_1 >$). According to our semantics, in the updated database $< f_3\ a_1\ c_1 >$ should not be present. From (1) it follows that $\neg(< f_1\ a_1\ b_1 > \wedge < f_2\ b_1\ c_1 >)$. At this juncture it is not known which of $\{< f_1\ a_1\ b_1 > , < f_1\ b_1\ c_1 > \}$ is false. To capture such partial information we employ three-valued logic. In this logic a fact can be "true", "false", or "ambiguous". Partial information is embodied by facts whose truth value is "ambiguous". An update results in changes in truth value of the relevant facts. In the above example, u_5 results in the following changes: $< f_3\ a_1\ c_1 >$ becomes false, $< f_1\ a_1\ b_1 >$ and $< f_2\ b_1\ c_1 >$ become ambiguous. That is, at least one of $\{< f_1\ a_1\ b_1 > , < f_2\ b_1\ c_1 > \}$ is false. This is represented by a construct called "negated-conjunction" (NC). The semantics of a NC are:

1. The conjunction of the facts in it is false.
2. Each fact in it is ambiguous.

Whenever a fact becomes a member of a NC its truth value is set to be ambiguous. The ambiguity of a fact is resolved directly by asserting the truth or falsity of this fact (through future inserts and deletes).

Next, consider the update u_6: INS(f_3, $< a_3, c_3 >$). Our semantics dictate that $< f_3, a_3, c_3 >$ holds in the updated database. The logical implication (2) indicates that there ∃ b such that $< f_1, a_3, b>$ and $< f_2, b, c_3 >$ are true in the updated database. In general, the specific value that b could take is not known. To accomodate this partial information we resort to null values [12]. Thus we will insert $< f_1, a_3, n_1 >$ and $< f_2, n_1, c_3 >$, where n_1 is a uniquely indexed null value. We call this chain of tuples the "null-valued chain" (NVC) of the derived fact $< f_3, a_3, c_3 >$.

The semantics of the truth value of facts are as follows. The truth values of base facts existing in the database are indicated by their logical state (true or ambiguous). Those not existing in the database are false. Derived facts do not exist in the database and their truth value is determined as follows. A derived fact can be obtained by composing a chain of base facts if adjacent pairs of facts in the chain match. Two facts $< x,y>$, $< u, v>$ match exactly if y=u, and match ambiguously if $y \neq u$ and (y is a null value or u is a null value)). Note that y = u iff both are non-null and y and u are the same data item, or both are null values with same index. A chain of base facts matches exactly if each adjacent pair of facts match exactly. A derived fact is true if it is obtained from a chain of true base facts which matches exactly. It is ambiguous if it can be obtained from a chain of base facts

[3]Base facts are tuples in the base relations: $< a_1, b_1> \in r_1$, $< a_1, b_2> \in r_1, \ldots, < c_1, d_1> \in r_3$

[4]Additionally u_4 also suggests $\neg(< a_1, b_2 > \in r_1 \wedge < b_2, c_1 > \in r_2 \wedge < c_1, d_1 > \in r_3)$.

which is not a superset of a NC and each chain of base facts from which it can be obtained either does not match exactly or contains at least one ambiguous fact. A derived fact is false if it is neither true nor ambiguous. According to these semantics a NVC implies a true derived fact if all its elements are true and an ambiguous derived fact if some of them are ambiguous.

In light of the above discussion we now provide semantics for the insert and delete operations in functional databases.

Semantics of Insert(σ)

1. σ is true.
2. A NC (σ, $\sigma_1, \ldots, \sigma_k$), and $\sigma \rightarrow (\sigma_1, \ldots, \sigma_k)$ is not a NC.
3. No other changes to the database.

An insertion of a fact is construed as an assertion of its truth. This is captured by (1). If σ is a base fact, (1) dictates that σ be physically stored in the database with a truth value true. In the case σ is a derived fact, it is represented by a NVC. Let σ be a conjunct of the NC (σ, $\sigma_1, \ldots, \sigma_k$). Since σ is asserted to be true now, it is not necessary that the conjunct $(\sigma_1, \ldots, \sigma_k)$ is false anymore. Thus it cannot be a NC. At the same time the truth value of any base fact other than σ itself should not be affected. This is captured by 3.

Semantics of Delete(σ)

1. σ is false.
2. A NC (σ, $\sigma_1, \ldots, \sigma_k$), and $\neg\sigma \rightarrow (\sigma_1, \ldots, \sigma_k)$ is not a NC.
3. No other changes to the database.

Deletion of a fact from the database is equivalent to asserting the falsity of the fact. This is stated in (1). If σ is a derived fact then its deletion is achieved by converting its derivations into NCs. If σ is a base fact we need to remove σ from the database extension. Let (σ, $\sigma_1, \ldots, \sigma_k$) be a NC. Since σ is a conjunct of it and σ is false now it is not necessarily true that $\{\sigma_1, \ldots, \sigma_k\}$ is a NC. This is captured by (2). However, the fact that σ_i, $1 \leq i \leq k$, is ambiguous is not affected by asserting $\neg\sigma$. This is ensured by (3).

4 Implementation of Update

A fact appearing in the database can be true or ambiguous. This information is stored along with the fact in the form of a truth-flag(set to T for true, or A for ambiguous). A fact can participate in the derivations of several derived facts. It is therefore possible for a fact to be a member of several NCs, and it is necessary to keep track of all the NCs that the fact is a member of. The "negated conjunction list" (NCL) attached to each fact maintains the set of NCs in which this fact participates. Now, a fact f(a) = b along with the relevant information is stored in the form of a quadruple < a, b, T/A, NCL> in the table corresponding to f.

The partial information arising from updates (on derived functions) is captured through the constructs: NC and NVC. Each NC has a unique index, and is implemented as a list of pointers to its component facts. In this way the NC and NCL form a dual data structure that enables the traversal from a NC to its component facts and vice versa.

4.1 Update Algorithms

In this subsection we will describe the update algorithms based on the above defined semantics. Before we present the actual update algorithms we will describe a set of operations on the two data structures – NC and NVC. A NC with index d is represented as NC(d). The following procedures define the set of operations on NCs.

```
Procedure create-NC(Conj-list);
% Conj-list is the list of conjuncts%
{generate NC with unique index, d;
for each element of Conj-list do
    { set its truth-flag to A;
    add d to its NCL;
    add element to NC(d) }}
```

```
Procedure dismantle-NC(d);
% Each element of NC(d) is ambiguous,%
% while their conjunction is not false. %
{ for each element p of NC(d) do
    { remove p from the NC(d);
    remove d from the NCL of p }
remove NC(d) }
```

The following procedures support the necessary operations on NVC.

```
Procedure create-NVC(f,x,y);
{ Let f = $f_1 \circ \ldots \circ f_k$;
Generate null values $n_{i_1}, n_{i_2}, \ldots, n_{i_{k-1}}$;
Add < x, $n_{i_1}$, T, nil> , < $n_{i_1}$, $n_{i_2}$, T, nil >
  ...< $n_{i_{k-1}}$, y, T, nil>
  to tables of $f_1, \ldots, f_k$ respectively }
```

```
Procedure clean-up-NVC(f,x,y);
% Make an ambiguous NVC true%
{ Let f = $f_1 \circ \ldots \circ f_k$
Let < x, $n_1$> < $n_1, n_2$ > ...< $n_{k-1}$, y>
    be the NVC of (f,x,y);
  base-insert ($f_1$,x,$n_1$);
  base-insert($f_2$,$n_1$,$n_2$);
  ⋮
  base-insert($f_k$,$n_{k-1}$,y) }
```

```
Function exists-NVC(f,x,y);
% Checks if NVC exists for derived fact%
{ Let f = $f_1 \circ \ldots f_k$;
if ∃ null-values $n_1, \ldots, n_{k-1}$ such that
  < x, $n_1$ > ∈ $f_1$, < $n_1, n_2$ > ∈ $f_2$;
    ...< $n_{k-1}$, y > ∈ $f_k$;
then return (yes);
else return (no) }
```

The following are update operations on base and derived functions. Observe that derived insert and derived delete cause partial information through the creation of NVC and NC, respectively. Base inserts and base deletes resolve the ambiguities of the facts by setting the truth flag and removing them, respectively.

```
Procedure base-insert(f,x,y);
{ if (< x, y > not in table of f) then
```

```
  add $< x, y, T, nil >$ to table of f;
  else { for each d in NCL of $< x, y >$ do
      dismantle-NC(d);
      Set the truth-flag of $< x, y >$ to T }
```

Procedure base-delete(f,x,y);

```
{ if ($< x, y >$ present in table of f) then
   { for each d in NCL of $< x, y >$ do
      dismantle-NC(d);
   remove $< x, y >$ from table of f }}
```

Procedure derived-insert(f,x,y);

```
{if (exists-NVC(f,x,y)) then
   clean-up-NVC(f,x,y);
   else create-NVC(f,x,y) }
```

Procedure derived-delete(f,x,y);

```
{ for each path p of (f,x,y) do
% p is a list of pairs deriving (f,x,y)%
      create-NC(p) }
```

4.2 Example

Let $\mathcal{F}$ be a functional database with the functions teach: faculty $\rightarrow$ course, class_list: course $\rightarrow$ student, and pupil: faculty $\rightarrow$ student. Pupil is a derived function with derivation teach o class_list. An instance of $\mathcal{F}$ is

```
Teach            | Class_list    | Pupil
-----------------|---------------|------------
euclid  math T {}|math john T {}|euclid  bill
laplace math T {}|math bill T {}|euclid  john
                 |               |laplace bill
                 |               |laplace john
```

Note that pupil is a set of implied facts, whereas class_list and teach are physically stored. Consider the following sequence of updates on $\mathcal{F}$.

u_1: DEL(pupil, < euclid, john>)
u_2: INS(pupil, < gauss, bill>)
u_3: DEL(teach, < euclid, math>)
u_4: INS(class_list, < math, john>)
u_5: INS(teach, < gauss, math>)

The result of u_1 is

```
Teach              | Class_list      | Pupil
-------------------|-----------------|--------------
euclid  math A {g1}|math john A {g1}|euclid  bill *
laplace math T {}  |math bill T {}  |laplace john *
                   |                 |laplace bill
```

We indicate ambiguous implied facts by a *. At this juncture $\mathcal{F}$ contains a NC, indexed by g_1, of the facts < teach, euclid, math> , and < class_list, math, john> .

The consequence of u_2 is

```
Teach              | Class_list      | Pupil
-------------------|-----------------|--------------
euclid  math A {g1}|math john A {g1}|euclid  bill *
laplace math T {}  |math bill T {}  |laplace john *
gauss   n1   T {}  |n1   bill T {}  |laplace bill
                   |                 |gauss   bill
                   |                 |gauss   john *
```

The effect of u_3 is

```
Teach            | Class_list     | Pupil
-----------------|----------------|-------------
gauss   n1   T {}|math john A {}|gauss   john *
laplace math T {}|math bill T {}|laplace john *
                 |n1   bill T {}|laplace bill
                 |               |gauss   bill
```

The result of u_4 is

```
Teach            | Class_list     | Pupil
-----------------|----------------|-------------
gauss   n1   T {}|math john T {}|gauss   john *
laplace math T {}|math bill T {}|laplace john
                 |n1   bill T {}|laplace bill
                 |               |gauss   bill
```

The outcome of u_5 is

```
Teach            | Class_list     | Pupil
-----------------|----------------|-------------
gauss   n1   T {}|math john T {}|gauss   john
laplace math T {}|math bill T {}|laplace john
gauss   math T {}|n1   bill T {}|laplace bill
                 |               |gauss   bill
```

Through the above example we explained how partial information is created by derived inserts (NVCs) and derived deletes (NCs). We have also seen how ambiguous information is resolved through deletes (falsifying ambiguous facts), and inserts (making ambiguous facts true).

In this section we have developed semantics for updating derived functions in functional databases. We have provided algorithmic procedures that perform updates according to these semantics. The relevant data structures to support these algorithms are also described.

5 Conclusion

The redundancy present in the specification of the conceptual schema poses problems to the maintenance of consistency of the database. This redundancy is in the form of derived functions. To ensure the consistency of the database it is necessary to identify all the redundancies and handle them appropriately. The identification of redundancy is achieved by obtaining the derived functions and their derivations. Off-line approaches [13] to perform the task of identifying derived functions rely upon constraints placed on the conceptual design. This renders the design process inflexible. Our goal is to provide a flexible design environment which enables the maintenance of the consistency of the databases so designed. We achieve this by developing an interactive design methodology.

The problem of updating derived functions is similar to the view update problem in relational databases. The crux of the problem is updates on views give rise to partial information. The solutions proposed by [6] [7] [8] [9], this partial information is approximated with various degrees of success. They measure the "goodness" of the approximation by its side effect on the database. We have advanced new update semantics based on definitions of derived functions. In our framework partial information is captured directly without being approximated. In this sense we achieve "side effect" free updates.

In any information processing system it is desirable to minimize the amount of ambiguous information. We have examined the resolution of ambiguities when additional information becomes available. It is clear that functional dependencies also play an important role in resolving partial information [12][5]. Other semantic constraints (integrity constraints, etc.) may also help resolve ambiguous information. In the presence of excessive ambiguous information it is desirable to quantify the degree of ambiguity. In this light the applicability of probabilistic and default logics must be investigated.

[5]In functional databases the type functional information indicates relevant functional dependencies.

References

[1] D. W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[2] R. S. Nikhil. *An Incremental, Strongly typed, Database Query language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 1984.

[3] K.G. Kulkarni and M.P. Atkinson. Efdm: extended functional data model. *The Computer Journal*, 29(1):38–45, 1986.

[4] E.F. Codd. Recent investigations in a relational database system. *Information Processing*, 1017–1021, 1974.

[5] A.L. Furtado, K.C. Sevcik, and C. S. Dos Santos. Permitting updates through views of databases. *Information Systems*, 4(4):269–283, 1979.

[6] U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(3), September 1982.

[7] J. E. Davidson. *Interpreting Natural Language Database Updates.* PhD thesis, Department of Computer Science, Stanford University, December 1983.

[8] A. M. Keller. *Updating Relational Databases Through Views.* PhD thesis, Department of Computer Science, Stanford University, February 1985.

[9] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *Proceedings of ACM Symp. of Princ. of Database Systems*, pages 352–365, Log Angeles, California, 1983.

[10] U. Dayal. *Schema Mapping Problems in Database Systems.* PhD thesis, Center for Research in Computing Technology, Harvard University, August 1979.

[11] A.P. Sheth, J.A. Larson, and E. Watlins. Tailor, a tool for updating views. In *Intl. Conf. on Extended Database Technology, appeared in Lecture Notes in Computer Science, No. 303*, pages 190–213, 1988.

[12] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[13] S. Lanka, S. Jha, and R. Yerneni. *Minimum Schema in a Functional Data Model.* Technical Report CS-88-07, Department of Computer Science, Penn State University, University Park, 1988.