Identifying and Update of Derived Functions in Functional Databases

Ramana Yerneni  Staram 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 specifications 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,

\[ F : \alpha \rightarrow \beta \]  \hspace{1cm} (1)

where \( F \) is the name of the function whose domain consists of objects of type \( \alpha \) and whose range consists of objects of type \( \beta \). 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\(^5\). For example, consider the conceptual schema S1 shown in table 1.

| 1. | grade: [student; course] \rightarrow letter.grade |
| 2. | score: [student; course] \rightarrow marks |
| 3. | cutoff: marks \rightarrow letter.grade |
| 4. | teach: faculty \rightarrow course |
| 5. | taught.by: course \rightarrow faculty |

\( S_1 \)

| Table 1: Conceptual Schema S1 |

\( S_1 \) 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 intentionally 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 \( \circ \) and is defined as \( x(f \circ g) = (xf)g \). Inverses of functions are defined as: \( f \) is the inverse of \( g \) if \( f = \{<a,b> | <b,a> \in g \} \).

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 \( cutoff \) (\( grade = score \circ cutoff \)). 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 \( grade \equiv score \circ cutoff \) 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
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 those 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>.

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 $F$ and $B$ is any subschema of $F$, then $B(I)$ denotes the set of functions of $B$ at $I$. A function $G$ is derived from $\{F_1, \ldots, F_n\}$ iff $\exists u_1, \ldots, u_n \in \{\text{identity, inverse}\}$ such that $G = u_1 f_1 \circ \ldots \circ u_n f_n$. A minimal schema of an FDB with schema $\mathcal{S}$ is the minimal subschema $\mathcal{M}$ such that $\mathcal{V}G \in \mathcal{S}$, either $G$ is in $\mathcal{M}$ or, $\exists F_{i_1}, \ldots, F_{i_n}$ in $\mathcal{M}$ such that $G$ is derived from $\{F_{i_1}, \ldots, F_{i_n}\}$.

The minimal schema problem (MSP) is stated as: given an FDB $F$, find a minimal schema of $F$. 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 cut-off 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 $G$ as

$$\mathcal{C} = \{g | g = U(f_i) \circ \ldots \circ U(f_n)\}$$

where $f_i \in \mathcal{G}, 1 \leq j \leq k, u_j \in \{\text{identity, inverse}\}$. $\mathcal{C}$ is a set of derived functions formed from $\mathcal{G}$ by composition and inverse. An instance $I$ of an FDB with schema $\mathcal{S}$ is in unique form if and only if $((\forall (h \in \mathcal{S}))((h) \subseteq \mathcal{G}))$ if and only if $f$ and $h$ are syntactically and type functionally equivalent implies $f$ and $h$ are semantically equivalent at $I$. The UFA for an FDB $F$ states that all possible instances of $F$ 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 $F$ with schema $\mathcal{S}$, the minimal schema of $\mathcal{F}$ in $\mathcal{S}$.

Proof: Assume otherwise. Let $\mathcal{M}$ be any minimal schema of $\mathcal{S}$ 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 $I$, if $f$ is $\mathcal{M}(I)$ 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 $F$ with schema $\mathcal{S}$ as an undirected graph $G = (V, E)$ where $V$ is the set of object types of $F$ (i.e., domains and ranges of the various functions) and $E = \{(D_1, D_2) | \text{for some } F \in S, F: D_1 \rightarrow D_2\}$. The syntax and type functionality of an edge follow from the function it represents. We define the syntax of a path $D_{i_1}, \ldots, D_{i_n}$ as $D_{i_1} \rightarrow D_{i_n}$. 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 $F$.

Output: Minimal schema $\mathcal{M}$ of $\mathcal{F}$.

Step 1: Construct $G_F$, the function graph of $\mathcal{F}$.

Step 2: $M = \emptyset$

For each edge $e \in E$ do

- if ($e$ a path $p$ in $G' = (V, E - M \cdot E \cdot p)$ such that $p$ is syntactically and type functionally equivalent to $e$)
Lemma that $H$ is syntactically and type functionally equivalent to $S$.

Proof: Let $M$ be the subchema outputed in step 3. Let $H \in S - M$. Since $H \notin M$, $\exists \ a \ path \ D_{i1}, \ldots, D_{ik} \ in \ G_M \ which \ is \ syntactically \ and \ type \ functionally \ equivalent \ to \ H$. This means $\exists \ a \ set \ of \ functions \ F_{i1}, \ldots, F_{ik} \ in \ M$ and $u_1, \ldots, u_{ik}$ such that $H$ is syntactically and type functionally equivalent to $u_1F_{i1} \ o \ \ldots \ o \ u_{ik}F_{ik}$. Under UFA this implies $\exists \ F_{i1}, \ldots, F_{ik} \ in \ M$ such that $H$ is derived from $\{F_{i1}, \ldots, F_{ik}\}$. Thus $H \in S - M$. $\therefore \ H \in S$.

Lemma 2: AMS correctly computes a minimal schema.

Proof: Let $M$ be the subchema outputed in step 3. Let $H \in S - M$. Since $H \notin M$, $\exists \ a \ path \ D_{i1}, \ldots, D_{ik} \ in \ G_M$ which is syntactically and type functionally equivalent to $H$. This means $\exists \ a \ set \ of \ functions \ F_{i1}, \ldots, F_{ik} \ in \ M$ and $u_1, \ldots, u_{ik}$ such that $H$ is syntactically and type functionally equivalent to $u_1F_{i1} \ o \ \ldots \ o \ u_{ik}F_{ik}$. Under UFA this implies $\exists \ F_{i1}, \ldots, F_{ik} \ in \ M$ such that $H$ is derived from $\{F_{i1}, \ldots, F_{ik}\}$. Thus $H \in S - M$. $\therefore \ H \in S$.

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

Proof: Step 1 and 3 are computed in $O(n)$ time. Step 2 consists of $n$ iterations. Each iteration involves a search traversal of the function graph which takes $O(n)$ 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 occurring conceptual schema cannot be admitted under the UFA. For example, consider the following conceptual schema S2.

- teach : faculty $\rightarrow$ course; (many -- many)
- class.list : course $\rightarrow$ student; (many -- many)
- lecturer.of : student $\rightarrow$ faculty; (many -- 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$^{-1}$ o 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$^{-1}$ o lecturer.of$^{-1}$) and class.list are semantically different. Also, teach and (lecturer.of$^{-1}$ o 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 syntactic 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 at most one cycle. This cycle can be identified in $O(2^l)$ time where $l$ is the length of the cycle. Thus method 4.1 takes $O(n^3)$ time (besides the dialogue with the designer). In 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.

\[
\begin{align*}
\text{teach} & : \text{faculty} \to \text{course}; \ (\text{many} - \text{many}) \\
\text{taught.by} & : \text{course} \to \text{faculty}; \ (\text{many} - \text{many})
\end{align*}
\]

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 derived function).

\[
\begin{align*}
\text{class_list} & : \text{course} \to \text{student}; \ (\text{many} - \text{many}) \\
\text{lecturer.of} & : \text{student} \to \text{faculty}; \ (\text{many} - \text{many})
\end{align*}
\]

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.

\[
\begin{align*}
\text{grade} & : \text{student; course} \to \text{letter.grade}; \ (\text{many} - \text{one}) \\
\text{attendance} & : \text{student; course} \to \text{attn.percentage}; \ (\text{many} - \text{one}) \\
\text{attendance.eval} & : \text{attn.percentage} \to \text{letter.grade}; \ (\text{many} - \text{one})
\end{align*}
\]

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.

\[
\begin{align*}
\text{score} & : \text{student; course} \to \text{marks}; \ (\text{many} - \text{one}) \\
\text{cutoff} & : \text{marks} \to \text{letter.grade}; \ (\text{many} - \text{one})
\end{align*}
\]

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.

\[
\begin{align*}
\text{taught.by} & = \text{teach}^{-1}; \ (\text{confirmed by the designer}) \\
\text{lecturer.of} & = \text{class_list}^{-1} \circ \text{teach}^{-1}; \ (\text{confirmed by the designer}) \\
\text{grade} & = \text{score} \circ \text{cutoff}; \ (\text{confirmed by designer}) \\
\text{grade} & = \text{attendance} \circ \text{attendance.eval}; \ (\text{invalidated by the designer})
\end{align*}
\]

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, < x1, y1, < x2, y2 >) 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 -> course, class_list: course -> student, and pupil: faculty -> student. Pupil is a derived function with derivation teach o class_list. An instance of F is \( \{ \text{teach} = \{ < \text{euclid, math} >, < \text{laplace, math} > < \text{laplace, physics} > \} \), class_list = \{ < \text{math, john} >, < \text{math, bill} > \} \}, \) pupil = \{ < \text{euclid, john} >, < \text{euclid, bill} >, < \text{laplace, john} >, < \text{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
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 derived facts. The most important operations 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: r1(AB), r2(BC), r3(CD), and the view v1(AD) with the following instance:

\[ v1 = \{<a1, b1>, <a1, b2>, <a2, c1>, <a2, c2>, <a3, c3>\} \]

It is ambiguous if it can be obtained from a chain of base facts in an order specified through data dependencies. Those not existing in the database are the same data item, or both are null values.

3.2 Update Semantics in Functional Databases

We perceive 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 \(F\) with the functions \(F1: A \rightarrow B, F2: B \rightarrow C,\) and \(F3: A \rightarrow C; f3\) is a derived function with derivation \(f1 \circ f2\). Note that the derivation is equivalent to the logical implications:

1. \(<f1, a, b> > <f2, b, c> \rightarrow <f3, a, c>\)
2. \(<f1, a, c> > b \rightarrow <f1, b, c>\)
3. An instance of \(F\) be \(\{<f1, a1, b1>, <f1, a2, b2>, <f2, b1, c1>, <f2, b3, c3>, <f3, a1, c1>, <f3, a2, c2>\}\). Consider an update \(u3, u4: DEL(f3, u3), INS(u4, u4)\) del information of the kind explained above. We propose new semantics in the framework of functional databases.
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($\sigma$)
1. $\sigma$ is true.
2. A NC ($\sigma_1, \ldots, \sigma_k$), 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 $\sigma$ is a base fact, (1) dictates that $\sigma$ be physically stored in the database with a truth value true. In the case $\sigma$ is a derived fact, it is represented by a NVC. Let $\sigma$ be a conjunct of the NC ($\sigma_1, \ldots, \sigma_k$). Since $\sigma$ 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 $\sigma$ itself should not be affected. This is captured by 3.

Semantics of Delete($\sigma$)
1. $\sigma$ is false.
2. A NC ($\sigma_1, \ldots, \sigma_k$), 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 $\sigma$ is a derived fact then its deletion is achieved by converting its derivations into NCs. If $\sigma$ is a base fact we need to remove $\sigma$ from the database extension. Let ($\sigma_1, \ldots, \sigma_k$) be a NC. Since $\sigma$ is a conjunct of it and $\sigma$ 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 $\sigma_i$, $1 \leq i \leq k$, 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 $<a, b, T/A, NCL>$. All the facts are stored in a database of the form of a quadruple $<x, y, T/A, NCL>$. The NCL can be updated using the following update algorithms.

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 $n_1, n_2, \ldots, n_{k+1}$;
add $<x, n_1, T, nil>$, $<n_1, n_2, T, nil>$
$\ldots<n_{k+1}, y, T, nil>$
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 $<x, n_1 \circ \ldots \circ n_k, T, nil>$
be the NVC of $(f,x,y)$;
base-insert($f_1,n_1$);
base-insert($f_2,n_2$);
$\ldots$
base-insert($f_k,n_k,y$)}

Function exists-NVC($f,x,y$);
% Checks NVC for derived fact%
{Let $f = f_1 \circ \ldots \circ f_k$;
if $\exists$ null-values $n_1, \ldots, n_{k-1}$ such that
$x, n_1 > \in f_1, < n_1, n_2 > \in f_2$
$\ldots<n_{k-1}, y > \in f_k$
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

115
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); Set the truth-flag of <2, y> to T } }

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 create-NC(p) }

4.2 Example

Let \( F \) be a functional database with the functions:\footnote{The functions teach: faculty \( \rightarrow \) course, \( \) class-list: course \( \rightarrow \) student, and pupil: faculty \( \rightarrow \) student. Pupil is a derived function with derivation teach \( \circ \) gauss.} Teach \( I \) Class-list \( I \) Pupil

\( \begin{array}{l}
\text{Teach} \\
\text{euclid math T} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{math john T} \\
\text{bill T}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{euclid bill} \\
\text{laplace bill}
\end{array} \)

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

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

The result of \( u_1 \) is

\( \begin{array}{l}
\text{Teach} \\
\text{euclid math A (g1)} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{math john A (g1)} \\
\text{bill T}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{euclid bill } \ast \\
\text{laplace bill}
\end{array} \)

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

The consequent of \( u_2 \) is

\( \begin{array}{l}
\text{Teach} \\
\text{euclid math A (g1)} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{math john A (g1)} \\
\text{bill T}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{euclid bill } \ast \\
\text{laplace bill}
\end{array} \)

The effect of \( u_3 \) is

\( \begin{array}{l}
\text{Teach} \\
\text{gauss ni T} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{ni bill T} \\
\text{laplace bill}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{gauss bill}
\end{array} \)

The result of \( u_4 \) is

\( \begin{array}{l}
\text{Teach} \\
\text{gauss ni T} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{ni bill T} \\
\text{laplace bill}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{gauss bill}
\end{array} \)

The outcome of \( u_5 \) is

\( \begin{array}{l}
\text{Teach} \\
\text{gauss ni T} \\
\text{laplace math T}
\end{array} \begin{array}{l}
\text{Class_list} \\
\text{ni bill T} \\
\text{laplace bill}
\end{array} \begin{array}{l}
\text{Pupil} \\
\text{gauss bill}
\end{array} \)

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 \cite{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 that updates on views can lead to partial information. The solutions proposed by \cite{6} \cite{7} \cite{8} \cite{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]. 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.

References


---

In functional databases the type functional information indicates relevant functional dependencies.