Assuring Retrievability from Unstructured Databases by Contexts

Amihai Motro

Department of Computer Science
University of Southern California
Los Angeles, CA 90089

Abstract

In an unstructured database the data is a collection of facts that does not adhere to any schema. Such a database does not require any initial design and can therefore evolve freely to accommodate new applications. It is particularly suitable for information which is diverse and idiosyncratic, such as when we want to store everything known on a particular topic. Unfortunately, this freedom also means that similar information may be entered in different forms. This may cause severe problems when retrieval is attempted, as some of the data may appear to have been "lost" in the database. In this paper we propose a method to solve this problem. Each database fact must be supported by a *context* in the database, in the form of several other facts. When an attempt is made to add a fact to the database, the existence of a suitable context is verified, or is extracted from the user in a simple dialogue. Thus, the database still retains the flexibility of unstructured databases, but problems of multiple representations are usually prevented.

1. Introduction

Most database management systems employ data models that are *structured* (or *strictly-typed*). The network, the hierarchical and the relational data models are all examples of the structured approach. Such models enforce a database design that is both *restrictive* and *permanent*. Restrictive, because the design relies heavily on broad categorizations, that apply to large classes of instances. Permanent, because in general these models require a priori commitment to a particular design. Consequently, structured models are suitable mostly for traditional database applications in which the environment to be modelled lends itself to simple categorizations and is relatively stable.

For example, a typical data model will record employees and departments with a fixed number of attributes, such as EMPLOYEE-NO, EMPLOYEE-NAME and EMPLOYEE-ADDRESS, DEPARTMENT-NAME, DEPARTMENT-HEAD and DEPARTMENT-OFFICE. The relationship between employees and departments will also have to be determined and defined; for example, WORKS-FOR may associate each employee with at most one department. These few generic attributes, that are applicable to all employees and all departments, are limited in their ability to capture the differences between individual instances of employees or departments. In addition, if this design later proves to be unsatisfactory, modifications may require substantial effort. While these limitations are not always objectionable, structured models are inadequate in situations where there is need to model data which is more diverse and idiosyncratic. An example is a database in which one wishes to record all that one knows about a topic. Such databases are quite impossible to design, as the data does not easily fit into uniform structures, and the eventual scope of the database is initially unknown.

An attractive approach for such situations is a database that is *unstructured* (or *loosely-typed*). The database is merely a container that can hold diversified information, into which one can toss information casually. Such an architecture requires no commitment to a particular design and can therefore accommodate any evolution in the contents of the database. As there is no structure, it can accommodate data with all its complexities and idiosyncrasies. A flexibility of this sort is available in *pile* structures, which are aggregates of records that do not adhere to any uniform record type and are not organized in any meaningful way (a detailed discussion of the applicability and performance of piles can be found in [17]). However, unstructured databases are not necessarily unorganized: to facilitate access they may adopt some internal organization, such as rings or indexes. Other efforts that can be classified as supporting an unstructured approach, are mostly based on semantic networks or logic (a good review of the topic can be found in [15]).

CH2261-6/86/0000/0426$01.00 © 1986 IEEE

In a structured database all individual updates are monitored rigorously and those that do not conform to its schema are rejected. Consequently, the data is always well organized, thus ensuring efficient retrieval later. An unstructured database accepts every update freely (as long as it does not introduce contradictions). This may create difficulties, when retrieval is attempted. Because composition of new facts is, for the most part, unrestricted, similar real world associations may find their way into the database as dissimilar facts. Consequently, because of these multiple representations, facts may appear to be "lost" in the database. One possible approach to this problem is to develop new retrieval methods to search for facts in unstructured databases. Such strategies, collectively referred to as browsing, were discussed in [12]. Note that most logic-based databases actually impose a relational structure by establishing a recognized set of predicates [16, 8, 4, 10]. Thus, in effect these are structured databases.

Our approach here is to require that each database fact be supported by a *context*, consisting of several other database facts. This requirement imposes connectivity on the collection of facts, thus preventing isolation of facts and decreasing the likelihood of multiple representations for similar facts. Contexts are also helpful as interpretations for database facts (Does "John likes Miller" represent affection towards a person or preference for a certain brand of beer?). In some sense, contexts substitute for a database schema, but can be enforced without sacrificing unstructuredness. Our approach is therefore termed *loosely structured*. The different approaches can be compared by the way they handle an unfamiliar update. A Loosely Structured database will neither reject it outright (as a structured database would), nor would it accept it casually (as an unstructured database would). Instead, it would check whether the database has an appropriate context to support this fact. If a context does not exist (or is incomplete) the missing facts will be obtained from the user through a simple dialogue. Thus, like an unstructured database the system will eventually accept every update; still, by guaranteeing a context, future retrievals will be as successful as with a structured database.

As an example, assume we want to add the fact that "John loves Mary" to the database. A structured database that has not anticipated this kind of information will reject it. An unstructured database will accept it, but if it already includes another fact, such as "Betty is-loved-by John", an eventual query, such as "Who does John love?", may not match both Betty and Mary. The system proposed here will extract from the user the equivalence of the two representations, and will add this information to the database as just another fact. This new equivalence fact will become part of the context of the update fact, and after it has been added, the update fact will be accepted. This context will guarantee success later.

Our model for databases is based on logic: a database is a collection of facts, monitored by a set of rules. But while most logic-based databases allow facts of arbitrary complexity, facts in our database are always binary. This approach has several advantages. First, binary representations are more "atomic", and therefore more suitable for modeling dynamic environments. For example, to add (or delete) a participant in an arbitrary fact requires that this fact be replaced by another fact of a different dimension. The binary approach handles this task by adding (or deleting) a single binary fact. Second, modeling idiosyncratic data is easier with binary facts, as complex facts tend to have limited applicability (contributing to the proliferation of "null values"). Third, this uniform representation tends to reduce the occurrence of different representations of the same information; as discussed earlier, such multiple representations are obstructive to successful retrieval. Note that these three advantages are all directly related to our purposes here. Finally, a database of binary facts can be viewed as a semantic network. This view can serve as the basis for convenient user interfaces. Admittedly, facts that are non-binary in their nature, must be coerced into binary representations. But as this is a purely mechanical operation, it presents no conceptual difficulties. Finally, we note that most of our methods can be extended without much difficulty to a model which allows facts of arbitrary complexity. The view of databases as systems of logic has been explored widely [7, 11]. Models based on a binary approach include [1, 3], and more recently [2].

Our model is similar to the one used in [12] and [13] and is summarized in Section 2. Section 3 then shows how to incorporate useful database features into this model. The concept of supporting context is discussed in Section 4, and Section 5 demonstrates its enforcement during updates. We conclude in Section 6 with a brief discussion of our results.

2. The Loose Structure Data Model

Our modeling approach is to view a real world environment as a collection of entities, which are related through facts, which in turn are monitored by rules. Our formalization is based on logic: a fact is a binary relationship between two entities; a rule is a formula to monitor the relationships among facts and to infer additional facts; a database is a contradiction-free system of facts and rules. In this section we define the basic model. Facts and rules can be used to incorporate into every database various features, which enhance its semantic capabilities. Such a "start-up kit" is described in the next section.

2.1. Entities and Facts

The basic unit of data is called *entity*, and we assume a universe $\mathcal{E}$ of distinctly named entities. This universe is partitioned into three sets, which are not necessarily disjoint: a set of *types* $\mathcal{T}$, a set of *tokens* $\mathcal{V}$ and a set of *relationships* $\mathcal{R}$. In general, types are entities that correspond to aggregates of instances (for example, PERSON, DEPARTMENT and SALARY), tokens are entities that correspond to individual instances (for example, JOHN, SHIPPING and $30000), and relationships are abstract entities which describe associations between other entities, whether types or tokens (for example WORK-FOR, EARN and LOVE). As an example of an entity which belongs to more than one set consider the entity BEER. It may correspond to a type aggregating all individual beers, and be an instance of another type representing all alcoholic drinks. Another example is the entity LOVE. It may represent both a relationship (e.g. between two persons) and a type (with particular instances, such as John's love to Mary).

Real world environments are represented in the database with binary associations between types or tokens, which are named with relationships. The resulting triplets of entities are called *facts*. Facts are therefore elements of the set $(\mathcal{T} \cup \mathcal{V}) \times \mathcal{R} \times (\mathcal{T} \cup \mathcal{V})$, and are considered the basic units of information. The first component of a fact will be referred to as the *source* of the fact, the middle component is the *relationship* of the fact, and the last component is the *target* of the fact. For example, that John loves Mary is represented in the database with the fact (JOHN,LOVE,MARY); JOHN is the source of this fact, MARY is the target and LOVE is the relationship. Our facts are similar to "points" in the cubic information space suggested by [2], and they recall the data structures used in LEAP [6].

This uniform treatment of entities, be they types, tokens or relationships, is similar to they way knowledge is represented in semantic networks, where each syntactic entity becomes a node and relationships between entities are expressed with labeled arcs (for example, [5, 14]).

2.2. Templates and Rules

A fact in which one or more entities are substituted for *variables* is called a *template fact*. A template fact is a restriction on its variables to entities that form existing database facts. An example of a template is $(x$,LOVE,MARY). The value of the variable x is restricted to entities which related to MARY through LOVE. Alternatively, this template is said to *match* all the facts with relationship LOVE and target MARY.

A *rule* is composed of a set of templates and an additional template. The set of templates represents a conjunction of bindings, which are then used to bind the additional template.

Rules are used to express *inference*. For example, the following rule inserts every student with GPA greater than 3.5 into the honor category:

$$(x,\in,\text{STUDENT})(x,\text{GPA},y)\ (y,>,3.5) \Rightarrow (x,\in,\text{HONOR-STUDENT}).$$

If the database includes the facts (JOHN,$\in$,STUDENT) and (JOHN,GPA,3.7), then, using this rule, the fact (JOHN,$\in$,HONOR-STUDENT) may be inferred[1] . An inference rule may therefore be regarded as a collective representation of facts. Rules can also be used to express *integrity constraints*. This is discussed in the next section.

2.3. Queries

A *query* is a formula constructed from template facts using negation, conjunction and disjunction operations, and universal and existential quantifiers. Let Q be such a query, and let $x_1,\ldots,x_n$ be its free variables. The *value* of Q is the set of tuples $(c_1,\ldots,c_n)$ that satisfy it.

For example, the query

$$Q(x) = (x,\in,\text{BOY}) \wedge ((\exists y)\ (y,\in,\text{GIRL}) \wedge (x,\text{BROTHER-OF},y)).$$

lists all boys who have a sister. For brevity, existential qualifications will be omitted in future queries.

3. A Database Start-Up Kit

In this section we describe particular facts and rules which implement several important database features. These facts and rules can be considered as a "start-up kit", which should be included in every database. Note that some of these features, included here as part of the *database*, are normally implemented as part of the *system*. One example is the so-called "semantic relationships", such as membership and generalization. Another example is "mathematical knowledge", such as the relationships between numbers.

3.1. Mathematical Facts

Often, a query may assert a mathematical relationship that the data must maintain. For example, "list all students with average grade under 2.5". To handle such queries we assume that the database includes all relevant mathematical relationships in the form of standard facts, with the appropriate mathematical comparator as relationship and the participating operands as source and target entities. Such facts are called *mathematical facts*. Examples are (2.2,<,2.5) and (30000,$\neq$,25000).

[1] The relationship $\in$ describes membership; it is discussed in the next section.

In particular, the set of tokens $\mathcal{V}$ should include all the numbers and set of relationships $\mathcal{R}$ should include the relationships $=$, $\neq$, $<$, and $>$. We assume that for every two different numbers N1 and N2 exactly one of the following facts is included: either (N1,<,N2) or (N1,>,N2), depending on whether N1 is smaller than N2 or not. In addition, we assume that for every two entities E1 and E2 (not necessarily numbers) exactly one of these two facts is included: either (E1,=,E2) or (E1,$\neq$,E2), depending on whether E1 and E2 are identical or not[2].

As an example, consider the previous query about students with average grade under 2.5. Its formal specification is:

$$Q(x) = (x,\in,\text{STUDENT})\ (x,\text{GPA},y)\ (y,<,2.5).$$

Assuming that the database includes the facts (JOHN,$\in$,STUDENT), (JOHN,GPA,2.4) and (2.4,<,2.5), the answer to Q will include JOHN.

3.2. Membership, Generalization and Consequence

The fundamental relationship between tokens and types is *membership*: a token is an instance of a type. To express this relationship with facts a special entity $\in$ is used. Facts in which $\in$ is the relationship are called *membership facts*. Example are (JOHN,$\in$,STUDENT) and (2.5,$\in$,NUMBER).

A frequent relationship between types is *generalization*: the concept described by one type is more general than the concept described by the other type. To express this relationship with facts a special entity $\prec$ is used. Facts in which $\prec$ is the relationship are called *generalization facts*. Example are (STUDENT,$\prec$,PERSON) and (GPA,$\prec$,NUMBER).

A third basic relationship is between relationships and it is called *consequence*: one relationship always implies another relationship between the same source and target. To express this relationship with facts a special entity $\Rightarrow$ is used. Facts in which $\Rightarrow$ is the relationship are called *consequence facts*. Examples are, (LOVE,$\Rightarrow$,LIKE) and ($<$,$\Rightarrow$,$\neq$).

The relations $\in$, $\prec$ and $\Rightarrow$ must be *disjoint* and their union must be *cycle-free*: two entities should not be related via more than one of these relationships, and an entity should not be related to itself through a chain of memberships, generalizations and consequences. In addition, it will prove useful to introduce three special entities, denoted TOKEN, TYPE and RELATIONSHIP which will be placed at the top of each hierarchy: every token will be a member of TOKEN, every type will be generalized by TYPE, and every relationship will be imply RELATIONSHIP. This can be enforced with appropriate inference rules.

The following rules express part of the semantics of membership, generalization and consequence:

$(x,\in,a)\ (a,\prec,b) \Rightarrow (x,\in,b)$,
$(x,r,y)\ (r,\Rightarrow,r') \Rightarrow (x,r',y)$,
$(a,\prec,b)\ (b,\prec,c) \Rightarrow (a,\prec,c)$, and
$(a,\Rightarrow,b)\ (b,\Rightarrow,c) \Rightarrow (a,\Rightarrow,c)$.

The first rule ensures that if a token is a member of a type, then it is also a member of every more general type. The second ensure that when two entities maintain a relationship, they also maintain every consequential relationship. The last two rules state the transitivity of generalizations and consequences.

3.3. Synonym

A frequent cause for failure of retrieval is the use of different database entities to represent the same real world entity. The same person may be represented as JOHN and JOHNNY, and the same relationship may be represented as TEACH and INSTRUCT. Consolidation of identical entities is possible if synonym information is included in the database. To express synonym relationships a special entity $\approx$ is used, and such facts are called *synonym facts*. Examples are (JOHN,$\approx$,JOHNNY) and (TEACH,$\approx$,INSTRUCT).

Synonym information can then be used to infer additional facts. An example of an inference rule is:

$(x,r,y)\ (r,\approx,r') \Rightarrow (x,r',y)$.

3.4. Inversion

Consider the facts (JOHN,LOVE,MARY) and (MARY,LOVED-BY,JOHN). By switching the source and target entities and using relationships that are "inverses" of each other, we obtain two different representations of the same information. If inverse information is available, then when given one representation, the other may be inferred. As inverse information is simply a relationship between entities, it may be stored in facts. To express inverse relationship, a special entity $\leftrightarrow$ is used, and such facts are called *inversion facts*. For example, (LOVE,$\leftrightarrow$,LOVED-BY).

Inference by inversion is then performed by the following rule:

$(x,r,y)\ (r,\leftrightarrow,r') \Rightarrow (y,r',x)$.

[2] Of course, as the number of such facts is infinite, their physical representation is more concise.

3.5. Contradiction

To monitor the consistency of the database we may include in the database information about relationships which are considered contradictory. Two relationships are contradictory if both may not be maintained between the same two entities. Such relationships may be mathematical (i.e. $=$ and $>$) or linguistic (i.e. LOVE and HATE). To express contradiction relationships a special entity $\perp$ is used, and such facts are called *contradiction facts*. Examples are $(=,\perp,>)$ and (LOVE,$\perp$,HATE).

Consider the following inference rule:

$$(x,r,y)\ (x,r',y)\ (r,\perp,r') \Rightarrow (\perp,\perp,\perp).$$

This rule detects contradiction by inserting into the database a special fact, called the *alarm fact*.

Contradiction information, together with this rule, can be used to express integrity constraints as rules of inference. For example, to state that every graduate student must maintain an average grade higher that 2.5, we define the rule:

$$(x,\in,\text{STUDENT})\ (x,\text{LEVEL},\text{GRADUATE})\ (x,\text{GPA},y) \Rightarrow (y,>,2.5).$$

If a graduate student has an average grade of 2.0, this rule will introduce the fact $(2.0,>,2.5)$. Assuming that the database already includes the facts $(2.0,<,2.5)$ and $(<,\perp,>)$, the previous rule will introduce the alarm, which should alert the system.

Synonym, inversion and contradiction are all symmetric relationships. With appropriate inference rules we can guarantee that indeed every such fact will appear in the database in its two forms.

3.6. Composition

Each fact describes a relationship from the source object of this fact to its target. When the target entity of one fact is the source entity of another fact, an indirect relationship between the source of the first fact and the target of the second fact is implied. For example, (JOHN,LOVE,MARY) and (MARY,DAUGHTER-OF,HARRY) imply a relationship between the entities JOHN and HARRY.

Let $p_1 = (s_1,r_1,t_1)$ and $p_2 = (s_2,r_2,t_2)$ be two facts. If $t_1 = s_2$ and $t_2 \neq s_1$ then p_1 and p_2 are said to be composeable and their *composition* is defined as the fact $(s_1,r_1.t_1.r_2,t_2)$, where $r_1.t_1.r_2$ is a new relationship entity composed from r_1, t_1 and r_2. In the above example, the composition of the fact (JOHN,LOVE,MARY) and fact (MARY,DAUGHTER-OF,HARRY) is the fact (JOHN,LOVE.MARY.DAUGHTER-OF,HARRY). Inference by composition is defined as follows:

$$(s,r,t)\ (t,u,v)\ (v,\neq,s) \Rightarrow (s,r.t.u,v).$$

Composition is a very powerful tool. In a database augmented with all composition facts, a template query such as (JOHN,x,MARY) will match all the composed relationships that relate John and Mary; e.g. HUSBAND-OF, FATHER-OF.NANCY.DAUGHTER-OF and WORKS-FOR.HARRY.FATHER-OF. Notice that by insisting that the source of the first fact is different from the target of the second fact, we avoid "cyclical" compositions. Otherwise, given (JOHN,LOVE,MARY) and (MARY,LOVE,JOHN), an infinite number of different composition facts would be generated; (JOHN,x,MARY) would then match an infinite number of different relationships.

4. Contexts

As noted earlier, without the restrictions of a schema, an unstructured database accepts almost any combination of entities for a fact (as long as it does not contradict existing facts). This contributes to multiple representations, and eventually to isolation of database facts. Our method to control these occurrences is to require that each fact be supported by a set of other database facts called a *context*. A context must satisfy an affiliation requirement for every entity of the fact, and an applicability requirement for the relationship of the fact.

4.1. Entity Affiliation

Our basic means for controlling the proliferation of database entities is by requiring *affiliation*. Each entity must be affiliated with at least one other entity. Token are affiliated with types through membership facts, types are affiliated with other types through generalization facts, and relationships are affiliated with other relationships through consequence facts. Thus, the hierarchies imposed on the entities by the relationships $\in$, $\prec$ and $\Rightarrow$ provide the fundamental form of connectivity.

Consider the fact (HARRY,LIKE,CAT). The entity HARRY may be affiliated with the entity PERSON through the fact (HARRY,$\in$,PERSON); LIKE may be affiliated with LOVE through (LOVE,$\Rightarrow$,LIKE); and CAT may be affiliated with MAMMAL through (CAT,$\prec$,MAMMAL). Note that if also (PERSON,$\prec$,MAMMAL), than by a previous inference rule (HARRY,$\in$,MAMMAL) is also a fact, establishing MAMMAL as another affiliate of HARRY.

4.2. Relationship Applicability

The affiliation requirement attempts to control individual entities. The applicability requirement is intended to control the arbitrary combination of entities into facts. For each fact that includes a token, the applicability of the particular relationship to the token must be supported by another fact in which the token is replaced by one of its affiliated types.

Consider again the fact (JOHN,LOVE,MARY). Assume that the affiliates of JOHN are STUDENT and PERSON; and that PERSON is also an affiliate of MARY. A fact such as (PERSON,LOVE,PERSON) will demonstrate the applicability of the relationship LOVE to JOHN and MARY. As another example, consider the fact (TOM,LIKE,BEER), where affiliations are provided by (TOM,$\in$,PERSON), (LIKE,$\Rightarrow$,RELATIONSHIP) and (BEER,$\prec$,DRINK). As TOM is the only token of this fact, applicability must be satisfied with (PERSON,LIKE,BEER).

Together, affiliation and applicability define the facts which are necessary to support every given fact. Such a set of facts is called a *context*. Thus, a set of facts is a context for a given fact if it provides all its entities with affiliations; and, unless it establishes both the source and the target of the given fact as types, it also supports the applicability of the relationship to the tokens of the fact.

5. Context Enforcement During Updates

The enforcement of contexts is part of the update mechanism of Loosely Structured databases. Each fact that is added must have a context, and each fact that is deleted must not deprive other facts from their only context. During insertion, if a context is not found (or if only a partial context is available) the system extracts the necessary context from the user in a simple dialogue. During deletion the system only checks that a fact can be deleted safely. As insertion is the more complex process we discuss it here.

To add a new fact to the database the system prompts the user thrice: for the source entity, for the relationship, and for the target entity.

First, the system retrieves the affiliations of the source entity. If none exist, the system asks the user what this entity is. If it is to be a token, the user answers by entering its intended type. The system then executes its *new token* procedure, as follows: if the type exists, the user is offered to examine other known instances of this type, as this helps avoid synonyms. If a synonymous entity is indeed discovered, it may be adopted in the new fact, or else an appropriate synonym fact is inserted. If the type specified by the user does not exist, the system executes its *new type* procedure (to be described later). Finally, the appropriate membership fact to affiliate the source token is inserted. If, on the other hand, the source entity is to be a type, the user answers by entering TYPE. The system then executes its *new type* procedure, as follows: it asks the user to examine existing types for synonyms or generalizations. If a synonymous type already exists, it may be adopted here, or else an appropriate synonym fact is inserted. If a more (or less) general type is detected, then an appropriate generalization fact is inserted. If the new type is not related to existing types, then a generalization fact is inserted which relates it to the most general type (TYPE), thus providing it with an affiliation.

After executing a similar process for the target entity, the system considers the relationship entity, trying to affiliate it and satisfy applicability. If the fact involves tokens, the system requires a fact with the same relationship, but with the tokens substituted with their affiliated types. If not found, the system executes its *new relationship* procedure, as follows: it asks the user to examine existing relationships *between the two types* for synonyms, inversions or consequences. If a synonymous or inverse relationship already exists, it may be adopted here, or else an appropriate synonym or inversion fact is inserted. If a consequence exists, an appropriate consequence fact is inserted. If these attempts are unsuccessful, a fact is inserted which makes the most general relationship (RELATIONSHIP) the consequence of the new relationship, thus providing it with an affiliation[3]. If the source and target of the given fact are types, the system only tries to affiliate the relationship. If no affiliation is found, then it simply executes its *new relationship* procedure.

Note that the process described above may involve "nesting" of dialogues, as it is possible that the user affiliates a new token with an unknown type. The system must then establish a context for this new type, before it continues with the update.

While a particular addition may involve substantial dialogue, others may be accepted instantly. Clearly, the first fact to describe a new kind of information will require that new types and relationships be established; additional facts of this kind are then absorbed more easily. The following example demonstrates a dialogue which is relatively long.

Assume a user wants to add the fact (CS101,TAUGHT-BY,TOM) to the database. The system detects that its source entity is affiliated with COURSE through the fact (CS101,$\in$,COURSE), but the target entity has no affiliations. Therefore it prompts the user with "What is TOM?", to which he answers PROFESSOR. Since PROFESSOR happens to be a known type, the user is then asked: "do you want to view other instances of PROFESSOR before adding TOM?" Assume he agrees and discovers that the same professor is already known to the database as THOMAS. The user now has the option of retaining TOM in his new fact, or substituting THOMAS

[3]When the user checks the new relationship against existing ones for a synonym, an inversion or a consequence, he may also discover a contradictory relationship. If pointed out, this information may be inserted into the database as a contradiction fact.

instead. If he retains TOM the system will insert the synonym fact (TOM,≈,THOMAS). Next, the system try to satisfy applicability by looking for the fact (COURSE,TAUGHT-BY,PROFESSOR) (finding it will also imply that TAUGHT-BY has affiliation). Assume it is not found, and the user is offered to examine the relationships from COURSE to PROFESSOR. When these reveal no synonymous relationships, he is offered to examine the relationships from PROFESSOR to COURSE. Here he identifies the relationship TEACH as the inverse of his own TAUGHT-BY. Again, he may be offered to abandon his formulation in favor of one that uses the existing relationship. As he declines, the inversion fact (TEACH,↔,TAUGHT-BY) is inserted. The context is now complete and the update is accepted. The context of (CS101,TAUGHT-BY,TOM) includes four facts:

(CS101,∈,COURSE),
(TAUGHT-BY,⇒,RELATIONSHIP),
(TOM,∈,PROFESSOR), and
(COURSE,TAUGHT-BY,PROFESSOR).

Note that two of these facts were *inferred*: the middle fact from (THOMAS,∈,PROFESSOR) and the synonym fact; the latter fact from (PROFESSOR,TEACHES,COURSE) and the inversion fact. The above update involved about half a dozen exchanges before the new fact was accepted. But if the user were now to insert the fact (CS201,TAUGHT-BY,TOM) it would be accepted instantly.

At various stages of the process the user is offered to examine sets of entities. Sets of types or relationships are relatively small, but sets of tokens (all members of a given type) may be larger. In general, however, token names are less likely to have synonyms than type names or relationship names. If the user is confident that the entity under consideration has a unique name these scannings can be by-passed.

6. Conclusion

Recently, there has been much interest in extending the capabilities of database management systems by incorporating into the system more "knowledge" about the data [9]. Often, "knowledge" is interpreted as generic information of a high level of abstraction, clearly distinct from "data", which are low level facts that apply to individuals [18]. This distinction seems to originate from the separation between data and schema that prevails in present database management systems.[4] Often, the models that separate knowledge from data intend to use the knowledge to control the data, and thus achieve databases that are more "intelligent". However, the knowledge base of the new system is in itself a valuable source of information, and there is no reason why it should not be available for query and update.

For this reason, while we may adopt the above definition of knowledge, we must be careful not to perpetuate the present dichotomy between data and schema, by separating data from knowledge. Instead, we should look for a model that treats information uniformly, and can express the *continuum* extending from simple facts to complex generalities. An essential part of this model should be flexible tools that implement a uniform approach towards retrieval and manipulation of all "levels" of information. This was achieved in part in the Loose Structure model we presented. Contexts may be regarded as "individual schemas" for each and every fact of an unstructured database. However, these "schemas" are simply collections of facts, stored and manipulated in the same way as all other facts. Thus, the model treats data and schema in a uniform and integrated way.

In this paper we concentrated on the update of facts in an unstructured model. Elsewhere we addressed the issues of retrieval in a very similar model [12, 13]. Still, in addition to facts, the system stores information in rules of inference (which are actually collective representations of facts). We must still develop query and update methods to manipulate this generic information with the same flexibility.

[4] Indeed, database schemas can be viewed as representation of knowledge (admittedly, limited) in present systems. They express generic information with classes, attributes, relationships and constraints.

References

[1] J. R. Abrial.
Data Semantics.
In J. W. Klimbie and K. L. Koffeman (editors), *Data Base Management*, pages 1-60. North-Holland, 1974.

[2] H. Afsarmanesh, D. Knapp, D. McLeod and A. Parker.
An Extensible Object-Oriented Approach to Databases for VLSI/CAD.
To appear in the Proceedings of the Eleventh International Conference on Very Large Data Bases.

[3] G. Bracchi, P. Paolini and G. Pelagatti.
Binary Logical Associations in Data Modelling.
In G. M. Nijssen (editor), *Modelling in Data Base management Systems*, pages 125-148. North-Holland, 1976.

[4] M. L. Brodie and M. Jarke.
On Integrating Logic Programming and Databases.
In *Proceedings of the First International Workshop on Expert Database Systems*, pages 40-62. Kiawah Island, South Carolina, 1984.

[5] A. Deliyanni and R. A. Kowalski.
Logic and Semantic Networks.
Communications of the ACM 22(3):184-192, 1979.

[6] J. A. Feldman and P. D. Rovner.
An Algol-Based Associative Language.
Communications of the ACM 12(8):439-449, August, 1969.

[7] H. Gallaire and J. Minker (editors).
Logic and Databases.
Plenum Press, 1978.

[8] M. Jarke, J. Clifford and Y. Vassiliou.
An Optimizing Prolog Front-end to a Relational Query System.
In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 296-306. Boston, Massachussets, 1984.

[9] L. Kerschberg (editor).
Proceedings of the First International Workshop on Expert Database Systems.
Kiawah Island, South Carolina, 1984.

[10] D. Li.
A Prolog Database System.
Research Studies Press Ltd. (John Wiley & Sons Inc.), 1984.

[11] H. Gallaire, J. Minker and J. -M. Nicolas.
Logic and Databases: A Deductive Approach.
Computing Surveys 16(2):153-185, June, 1984.

[12] A. Motro.
Browsing in a Loosely Structured Database.
In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 197-207. Boston, Massaccusetts, 1984.

[13] A. Motro.
Query Generalization: A Technique for Handling Query Failure.
In *Proceedings of the First International Workshop on Expert Database Systems*, pages 314-325. Kiawah Island, South Carolina, 1984.

[14] J. Mylopoulos and H. J. Levesque.
An Overview of Knowledge Representation.
In M. L. Brodie, J. Mylopoulos and J. W. Schmidt (editors), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Lanuages*, chapter 1. Springer-Verlag, 1984.

[15] D. C. Tsichritzis and F. H. Lochovsky.
Data Models.
Prentice Hall, 1982.

[16] D. H. D. Warren.
Efficient Processing of Interactive Relational Database Queries Expressed in Logic.
In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 272-281. Cannes, France, 1981.

[17] G. Wiederhold.
Database Design (Second Edition).
McGraw-Hill, 1983.

[18] G. Wiederhold.
Knowledge and Database Management.
IEEE Software 1(1):63-73, January, 1984.