Browsing in a Loosely Structured Database

Amihai Motro

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

Abstract

Current database architectures emphasize structure and are inappropriate for applications which model environments that are subject to constant evolution, or environments which do not lend themselves to massive classifications. In this paper we describe an architecture which promotes databases that are only *loosely structured*: heaps of facts instead of highly structured data. This architecture avoids the traditional dichotomy between "schema" and "data", and it incorporates a single mechanism for defining both inference rules and integrity constraints. As lack of organization will usually have adverse effect on retrieval, the principal retrieval method for loosely structured databases is *browsing*: exploratory searching which does not assume any knowledge of the organization (or even the very existence of organization). Two styles of browsing, called navigation and probing, are defined. Both are derived from a standard query language based on predicate logic.

1. Introduction

Database work may be divided into two types. On one hand there is *organization*: this includes the design and construction of the database, as well as its continuous maintenance (update and reorganization). On the other hand there is *utility*: the retrieval of data from the database. These two types of effort involve a natural trade-off: investment in organization is compensated by convenient and efficient retrieval. A simple example is a sequential file. Keeping it sorted is an investment, which yields benefits when the file has to be searched.

In an attempt to achieve efficient access, all current database methodologies require substantial investment in "appropriate" organization. This usually implies that databases are designed as highly structured aggregates of data. The general approach is to use low-level units of description such as entity types and relationship types, which are data structures of some kind; the global view of a modelled environment is then provided by a database schema, which is a structured aggregate of lower level types.

The alternative approach is to invest less in organization. We can imagine a database which is merely a "heap" of facts, without adherence to any "schema". Such an architecture requires no design and very little effort in maintenance. While the organized approach to databases is appropriate for many applications, there are situations in which the alternative approach may be more beneficial. Clearly, if the anticipated utility is low, investment in organization may be unjustified. Also, if the database is of relatively small size, the benefits of efficient retrieval may not be enough to warrant organization. Still, the greatest disadvantage of structured architectures is their rigidity. Every database is designed to model a particular environment. In doing so it actually "freezes" an environment, which may be evolving continuously. To assure that the database presents a faithful model of an environment which is evolving, its structures have to undergo similar evolution. (The same situation exists when the environment is stable, but our perception of it undergoes evolution.) Unfortunately, continuous evolution of database structures is virtually impossible (and periodic adjustment, usually termed restructuring, is very difficult and costly). We argue that structural evolution is simpler in architectures that do not emphasize structure. A related difficulty is encountered in applications that span more than one database. Because of the independent database structures involved, it is usually difficult to provide such applications with a single global view of their data. Unified access to multiple databases is much simpler with databases whose architecture does not emphasize structure.

According to the trade-off principle mentioned above, lack of organization will have adverse effect on retrieval. This calls for a different approach to retrieval as well. Standard retrieval mechanisms are designed to retrieve data from organized aggregates. Furthermore, most query languages require the user to know how the data is organized. Whether the user navigates in a network database, or expresses a query in relational algebra, some knowledge of the database schema is essential. In contradistinction, the appropriate mechanism for retrieval in unorganized environments is *browsing*: exploratory searching which does not assume any knowledge about the organization of the environment searched (or even the existence of such an organization).

Representation of information as a collection of facts is a method that has been employed in knowledge representation, in the form of semantic networks (see [8] for a recent review of this topic). The programming system PROLOG [4] also uses such collections of facts as part of the deduction systems that constitute "programs". More generally, the view of databases as systems of logic has been explored widely (for example, [5, 10]). Still, the use of unstructured collections of facts for databases has been limited, partly because of the notation of logic which is considered unattractive to database practitioners, but mostly because of the prevailing sentiment that a data model should enforce homogeneity [12].

One simplifying feature, that may help a logic-based approach overcome some of the difficulties that such approaches had traditionally encountered, is to combine it with a *binary* view of data. Models based on binary relationships have gained considerable attention because of their simplicity and intuitive appeal [1, 3, 12]. While most "logical databases" allow facts of arbitrary complexity, we propose that facts be limited to named pairs (triplets). These triplets are similar to "points" in the cubic information space suggested in 3DIS [2]. Because a database of such facts may be regarded as a collection of several binary relations, such an architecture may also be classified as "binary". An architecture based on binary facts combines the simplicity of binary models with the advantages found in the unstructured environment provided by logical models.

The framework of a logical model with binary facts is natural for the specification of the so-called "semantic relationships", such as membership and generalization. Semantic relationships have been incorporated successfully into systems like TAXIS [9] and SDM [6], which require that their data is structured according to a "schema", and offer them as "built-in" modelling tools. In the architecture we propose such relationships are not treated in a preferred manner. Rather, the individual characteristics of particular relationships are expressed in rules of inference and integrity.

The usefulness of database browsing as an alternative retrieval method has already been recognized, and sophisticated systems such as SDMS [7] and TIMBER [11] are either available or under construction. However, these browsers are designed to be used in conjunction with databases that are organized (the organization in both systems is relational). Consequently, before browsing in a virtual TIMBER relation or in a new SDMS data surface, the user must perform standard retrievals; such retrievals require knowledge of the organization of the database. Thus, a user who wishes to find out "something interesting" about John, cannot use such browsers effectively, as he must first know all the relations where the token *JOHN* may be found (or an extensive scan will be required).

The architecture we propose includes a "standard" query language based on predicate logic. Browsing is then available in two different styles. *Navigation* is an iterative process in which a user examines the "neighborhood" of a fact, picks a fact from this neighborhood, examines its neighborhood, and so on. It is intended for users who do not know what to look for, or do not know enough about the database to formulate a standard query. Navigation is effected by using a restricted type of queries, and can therefore be interleaved with standard queries. *Probing* is a querying mode in which every failure of a query is interpreted as "overqualification" of the target data. In this mode every failure initiates a set of "retraction" queries that attempt to broaden the target of the query. These retraction queries may provide satisfactory answers to the user's needs, or they may give him better understanding of the failure. Probing is intended to assist the browser who attempts standard queries, on the basis of limited familiarity with the database.

In this paper we describe a model for a database architecture that deemphasizes structure, while relying on browsing as its principal retrieval method. In the first part (Sections 2 and 3) we define the concept of a loosely structured database. The second part (Sections 4 and 5) is devoted to the browsing mechanism. In Section 6 we propose some additional mechanisms to facilitate the use of the system, and we conclude with a brief summary and problems that require further research.

2. A Loosely Structured Database

Most data models encourage classification. Prevailing wisdom is that the designer should attempt to make such observations about the environment, that capture a relatively high number of facts. Thus, a relation with many tuples is "justified", while a relation with a single tuple is "wrong". The data model that we describe does not insist on such universal classifications. Rather, it formalizes the notion of a basic unit of information (called

fact), and allows one to describe such units "one by one". In a sense, it is not a real data model, because it does not encourage "modelling", as this activity is usually understood.

2.1. Entities and Facts

Databases are models of real world environments, constructed in accordance with a set of modelling rules. Our modelling rules are extremely simple. We view a real world environment as a collection of entities, which are related via facts.

Entities are the basic units of data. Examples of entities are *JOHN* (a particular person), *PERSON* (an abstract entity) and *$25000* (a particular amount of money). We assume a universe $\mathcal{E}$ of distinctly named entities.

Real world entities may be associated in many different ways. These associations are represented in the database as binary relationships between the corresponding database entities. For example, the database entities *JOHN* and *$25000* may be related to represent the fact that the annual salary of John is $25000. However, it may be the case that the total debt of John is also $25000. To distinguish among different relationships between the same two entities, we name each relationship with an additional entity. A named pair of entities is called a *fact*. The set of entities that are used for relationship names is a subset of the universe of entities $\mathcal{E}$ and will be denoted $\mathcal{R}$. Facts are therefore elements of the set $(\mathcal{E} \times \mathcal{R} \times \mathcal{E})$, and are considered the basic units of information. Examples of facts are *(JOHN,EARNS,$25000)* and *(JOHN,OWES,$25000)*.

It is convenient to name the different positions in a fact triplet: the first and last positions are called *source* and *target*, respectively; the middle position is called *relationship*. Thus, the fact *(s,r,t)* states that the entity *s* is related to the entity *t* via the relationship *r*.

2.2. Types of Relationships

Consider the facts *(EMPLOYEE,EARNS,SALARY)* and *(EMPLOYEE,TOTAL-NUMBER,120)*. While both *EARNS* and *TOTAL-NUMBER* are relationships that apply to *EMPLOYEE*, there is a fundamental difference in their meaning: *EARNS* is an attribute of *EMPLOYEE* because it is characteristic of every individual employee; *TOTAL-NUMBER* is an attribute of *EMPLOYEE* because it characterizes the aggregate of all employees.

Relationships which characterize an entity because they are applicable to every instance of this entity, are called *individual* relationships; relationships which characterize an entity, but are not applicable to every instance of this entity, are called *class* relationships. This distinction partitions the set of all relationships into $\mathcal{R}_i$, the set of all individual relationships, and $\mathcal{R}_c$, the set of all class relationships.

2.3. Generalization and Membership

A frequent relationship between entities is that of *generalization*: the concept described by one entity is more general than the concept described by the other entity. To express this relationship with facts a special entity $\prec$ is used. Facts in which $\prec$ is the relationship are called *generalization facts*. Examples of generalization facts are *(EMPLOYEE,$\prec$,PERSON)* and *(SALARY,$\prec$,COMPENSATION)*. Note that relationship entities (that is, elements of $\mathcal{R}$) may also be related via generalization; for example, *(LOVES,$\prec$,LIKES)*. Generalization is an individual relationship, and it is reflexive: for each $E \in \mathcal{E}$: $(E,\prec,E)$ is in the database.

Generalization facts may be regarded as imposing a (partial) *hierarchy* on the entities of the database. It is sometimes convenient to assume the existence of two special database entities: one at the top of the hierarchy, the other at its bottom. We shall denote them with Δ and ∇, respectively. They maintain:

$\forall E \in \mathcal{E}$: $(E,\prec,\Delta)$ *and* $(\nabla,\prec,E)$

The entity Δ is the generalization of every other entity, and therefore is the most abstract entity. The entity ∇ is generalized by every other entity, and therefore is the most specified entity.

Another frequent relationship is that of *membership*: one entity is an instance of another entity. To express this relationship with facts a special entity $\in$ is used. Facts in which $\in$ is the relationship are called *membership facts*. Examples of membership facts are *(JOHN,$\in$,EMPLOYEE)* and *($25000,$\in$,SALARY)*. Note that an entity which is an instance of another entity (i.e. a source of a membership fact) may still have its own instances (i.e. may still be the target of membership facts). For example, the entity *ISBN-914894* is an instance of the entity *BOOK*, and has instances *ISBN-914894-COPY1* and *ISBN-914894-COPY2*. Membership is a class relationship.

2.4. Inference

Occasionally, a fact may be "represented" in the database although not specified explicitly. Such facts are called inferred facts. Inference of new facts is performed in accordance with certain rules.

If John is an employee and employees earn salaries, we would conclude that John earns a salary. Therefore, if the first two assumptions are captured by *(JOHN,$\in$, EMPLOYEE)* and *(EMPLOYEE,EARNS,SALARY)*, we would like to conclude that *(JOHN,EARNS,SALARY)* is

also a fact. If Tom is another employee, we would conclude that he too earns a salary. In general we can express these conclusions with the following inference rule:

(x,∈,EMPLOYEE), (EMPLOYEE,EARNS,SALARY)
⇒ (x,EARNS,SALARY).

x is a *variable*. Facts that include variables are called *templates*. Continuing this example, if the relationship *EARNS* is replaced by some other individual relationship, or the entity *SALARY* is replaced by some other entity, analogous inferences could still be made. A more general inference rule would be:

$\forall r \in \mathcal{R}_i$: *(s,∈,t), (t,r,u) ⇒ (s,r,u).*

Particular inference rules suitable for databases are discussed in Section 3.

2.5. Integrity

To ensure that the data in the database represents the environment it models accurately, we can state various relationships which the data must maintain. For example, we may wish to state that age cannot have a negative value, or that the salary of an employee never exceeds the salary of his manager. These required relationships can be stated with rules like:

(x,∈,AGE) ⇒ (x,GREATER,0), or
(x,∈,EMPLOYEE), (y,∈,EMPLOYEE),
(u,∈,SALARY), (v,∈,SALARY),
(x,EARNS,u), (y,EARNS,v),
(y,MANAGER,x) ⇒ (v,GREATER,u).

Such rules, called *integrity constraints*, are identical to inference rules, as they point out additional facts that must be present in the database. However, to preserve the meaning of such constraints we would like to disallow facts like *(2,GREATER,3)*. Assuming that the database already contains all valid mathematical relationships, including *(2,SMALLER,3)*, the presence of *both* facts constitutes a contradiction.

Two facts *(x,r,y)* and *(x,r',y)* are said to be *contradictory* if *r* and *r'* represent two relationships that cannot both be maintained between *x* and *y* at the same time. As we shall see in the next section, information on contradictory relationships is also expressed with facts.

2.6. Database

Collectively, inference rules and integrity constraints are referred to as *rules*. Although more general rules may be introduced, in this paper we assume that rules are strictly conjunctive: one set of templates implies another set of templates. Each rule may therefore be specified with two sets of templates:

Let $\mathcal{V}$ be a set of entity variables and denote $\mathcal{EV} = \mathcal{E} \cup \mathcal{V}$ and $\mathcal{RV} = \mathcal{R} \cup \mathcal{V}$. A *rule* is a pair $<L,R>$, where $L, R \subset (\mathcal{EV} \times \mathcal{RV} \times \mathcal{EV})$.

Given a set of facts $\mathcal{P}$ and a set of rules $\mathcal{R}$, the set of facts that may be obtained by repeated application of the rules in $\mathcal{R}$ to the facts in $\mathcal{P}$ is called the *closure of $\mathcal{P}$ under $\mathcal{R}$*. Clearly, every closure of $\mathcal{P}$ includes $\mathcal{P}$ itself.

Finally, a *loosely structured database* is a set of facts $\mathcal{P}$ and a set of rules $\mathcal{R}$, such that the closure of $\mathcal{P}$ under $\mathcal{R}$ is free of contradictions. As the set of rules is relatively stable, database construction involves primarily the specification of its facts.

As the definitions suggest, we regard databases as relatively unstructured: any "random" contradiction-free collection of facts qualifies as a database. In particular, note that we permit the following situations: the same two entities may be related through different relationships (e.g. *(MARY,MAJOR,MATH)* and *(MARY,ASSISTANT, MATH)*); the same relationship may exist between a variety of entity pairs (e.g. *(JOHN,LIKES,FELIX)* and *(PERSON,LIKES,PERSON)*); many to many relationships between entities are possible (e.g. *(TOM,ENROLLED-IN,CS100), (TOM,ENROLLED-IN, MATH101)* and *(SUE,ENROLLED-IN,MATH101)*); even "inconsistencies" and "replications" are allowed (e.g. *(JOHN,EARNS,$25000), (JOHN,EARNS,$40000)* and *(JOHN,INCOME,$40000)*).

We have selected the fact, which is a named relationship between two entities, as the atomic unit of information. Often, information may appear as a more complex relationship. For example, "Tom is enrolled in the CS100 and received the grade A" involves a relationship between three entities: *TOM*, *CS100* and *A*. In this case we must break down this complex fact into three atomic facts, using a new entity to identify the particular enrollment in question, say *E123*. Together, the three facts *(E123,ENROLL-STUDENT,TOM), (E123,ENROLL-COURSE,CS100),* and *(E123,ENROLL-GRADE,A)* represent the complex fact adequately.

An interesting feature of this data model is that it avoids traditional differences between schema and data. The concept of a fact results in a unified storage strategy for both "schema relationships" (e.g. *EMPLOYEE - INCOME*) and "data relationships" (e.g. *JOHN - $25000*). The retrieval mechanism we now describe provides a unified access strategy of "schema" and "data" as well.

2.7. Retrieval

Variables and templates were introduced to express inference rules and integrity constraints. They are also instrumental in the retrieval mechanism. When presented

to a database, a template acts as a query: it evaluates to all the facts in the database closure that match its non-variable components. In a database about books, the template *(y,∈,BOOK)* evaluates to the set of all facts that have ∈ as their relationship and *BOOK* as their target. In other words, it evaluates to the set of all books. Another example is *(x,y,z)*. Its value is, of course, the complete closure of the database.

Assume the database about books stores citations with facts of the type *(BOOK,CITES,BOOK)*. The template *(x,CITES,y)* matches all citation facts. To match self-citations only (books that reference themselves), we need to specify that x should be equal to y: *(x,CITES,x)*. This simple example demonstrates the need to express additional constraints on the variables. As a more complex example, assume that information on authorship is also available via facts of the type *(BOOK,AUTHOR,PERSON)*. To match all authors that cite themselves, we need to express the query "all authors y for which there exists a book x and *(x,CITES,x)* and *(x,AUTHOR,y)*". In general, each template is a predicate, which is satisfied (evaluates to *true*) if it matches a non-empty set of database facts. A query is then a formula constructed from such predicates using conjunction and disjunction operations, and universal and existential qualifiers. To obtain a formal definition of a query we begin with the definition of a (well-formed) formula:

1. Templates are the only predicates, and each predicate is an atomic formula.

2. If $\mathcal{A}$ and $\mathcal{B}$ are formulas and x is a variable, then $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, $((\forall x)\,\mathcal{A})$ and $((\exists x)\,\mathcal{A})$ are formulas.

3. An expression is a formula only if it can be shown to be a formula on the basis of the previous two clauses.

In $(\forall x)\mathcal{A}$ and $(\exists x)\mathcal{A}$, $\mathcal{A}$ is called the *scope* of the quantifier. An occurrence of the variable x is *bound* in a formula, if it is within the scope of $(\forall x)$ or $(\exists x)$ (or is the variable itself of one of these quantifiers). Otherwise, it is said to be *free* in the formula. A formula $\mathcal{Q}$ with free variables $(x_1,x_2,\ldots,x_n)$ is *satisfied* by the tuple $(c_1,c_2,\ldots,c_n)$, if $\mathcal{Q}(c_1,c_2,\ldots,c_n)$ is true.

Finally, let $\mathcal{Q}(x_1,x_2,\ldots,x_n)$ be a formula with $x_1,x_2,\ldots,x_n$ as its only free variables. $\mathcal{Q}$ is called a *query*. Its *value* is the set of all tuples $(c_1,c_2,\ldots,c_n)$ which satisfy it: $\{ (c_1,c_2,\ldots,c_n) : \mathcal{Q}(c_1,c_2,\ldots,c_n) \}$. The value of query $\mathcal{Q}$ is denoted $\{\mathcal{Q}\}$.

The previous query is now expressed with the following formula:

$$\mathcal{Q}(y) = (\exists x)\,((x,\in,BOOK) \wedge (y,\in,PERSON) \wedge (x,CITES,x) \wedge (x,AUTHOR,y)).$$

A formula without any free variables (closed formula) represents a proposition. An example of a proposition query is:

(JOHN,LIKES,FELIX) ∧ *(FELIX,LIKES,JOHN)*.

It is true if John and Felix like each other, and false otherwise.

Note that the query language does not include a logical negation operator. This should not present difficulties, as negative assertions can be made using a complementary relationship. For example, the query "all books whose author is not John" is expressed with the formula:

$$\mathcal{Q}(x) = (\exists y)\,((x,\in,BOOK) \wedge (y,\in,PERSON) \wedge (x,AUTHOR,y) \wedge (y,\neq,JOHN)).$$

This formal retrieval language is quite powerful. In this language, templates are a restricted type of queries. Such primitive queries will prove to be useful when we define our browsing mechanisms.

3. Standard Inference Rules

Inference and integrity rules provide concise representation for additional facts that, although not stored, are part of the database. While database integrity rules are particular to the environment modelled, database inference rules are more universal. In this section we discuss several inference rules appropriate for databases.

3.1. Inference by Generalization

Assume an organization where every employee works for some department, and some employees are managers. An obvious conclusion is that in this organization every manager works for some department. That is, given the facts *(EMPLOYEE,WORKS-FOR,DEPARTMENT)* and *(MANAGER ,≺,EMPLOYEE)* we can infer the fact *(MANAGER,WORKS-FOR,DEPARTMENT)*. If every employee earns a salary, and salary is some kind of compensation, then every employee earns compensation. That is, from the facts *(EMPLOYEE,EARNS,SALARY)* and *(SALARY,≺,COMPENSATION)* we can infer the fact *(EMPLOYEE,EARNS,COMPENSATION)*. Finally, assume that work for an employer implies payment by this employer. This assumption may be expressed with the generalization fact *(WORKS-FOR,≺,IS-PAID-BY)*. Consequently, if John works for the Shipping department, he gets paid by the Shipping department. That is, the facts *(JOHN,WORKS-FOR,SHIPPING)* and *(WORKS-FOR,≺,IS-PAID-BY)* imply the fact *(JOHN,IS-PAID-BY, SHIPPING)*.

These examples demonstrate the inferences possible with generalization facts. Formally, they are stated with the following inference rules:

$$\begin{aligned} &\forall r \in \mathcal{R}_i\text{: } (s,r,t) \text{ and } (s',\prec,s) \Rightarrow (s',r,t),\\ &\forall r \in \mathcal{R}_i\text{: } (s,r,t) \text{ and } (r,\prec,r') \Rightarrow (s,r',t),\\ &\forall r \in \mathcal{R}_i\text{: } (s,r,t) \text{ and } (t,\prec,t') \Rightarrow (s,r,t'). \end{aligned} \qquad (1)$$

By selecting $\prec$ for the relationship r in either the first or the last rule, we obtain the transitivity of generalization relationships:

(s,≺,t) and (t,≺,t') ⇒ (s,≺,t').

3.2. Inference by Membership

To continue the previous example, assume now that John is an employee and that every employee works for some department. Clearly, we can conclude that John works for some department. That is, the fact *(JOHN,WORKS-FOR,DEPARTMENT)* can be inferred from the facts *(JOHN,∈,EMPLOYEE)* and *(EMPLOYEE, WORKS-FOR,DEPARTMENT).* If Tom works for Shipping, and Shipping is a department, then we can conclude that Tom works for some department. That is, from *(TOM,WORKS-FOR,SHIPPING)* and *(SHIPPING,∈,DEPARTMENT)* we can infer the fact *(TOM,WORKS-FOR,DEPARTMENT).*[1]

Formally, inference by membership is stated as follows:

$\forall r \in \mathcal{R}_i$: *(s,r,t) and (s',∈,s) ⇒ (s',r,t),*

$\forall r \in \mathcal{R}_i$: *(s,r,t) and (t,∈,t') ⇒ (s,r,t').* (2)

By selecting $\prec$ for the relationship r in the first rule, we obtain:

(s',∈,s) and (s,≺,t) ⇒ (s',∈,t).

That is, if one entity is an instance of another entity, then it is also an instance of every more general entity.

3.3. Synonym Facts

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 both as *JOHN* and *JOHNNY*, and the same relationship may be represented as *SALARY*, *WAGE* and *PAY.* 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,≈,JOHNNY), (SALARY,≈,WAGE)* and *(SALARY,≈,PAY).* Recall that *(s,≺,t)* expresses the fact that s is a kind of t. Clearly, if *(t,≺,s)* also holds, i.e. t is also a kind of s, then s and t are synonyms. This provides us with a definition of the synonim relationship:

(s,≈,t) ⇔ (s,≺,t) and (t,≺,s).

Synonim information is used to infer additional facts, according to the following rule:

Given (r,≈,r'), r may be replaced with r' in every fact. (3)

For example, given the facts *(JOHN,EARNS,$25000)* and *(JOHN,≈,JOHNNY)* the fact *(JOHNNY,EARNS, $25000)* may be inferred. The symmetry of the synonim relationship is obvious from its definition. The reflexivity and transitivity of the synonim relationship are guaranteed by the reflexivity and transitivity of the generalization relationship. Using symmetry and transitivity we can now infer the synonym *(WAGE,≈,PAY)* from *(SALARY,≈,WAGE)* and *(SALARY,≈,PAY).*

3.4. Inversion Facts

Consider the facts *(INSTRUCTOR,TEACHES, COURSE)* and *(COURSE,TAUGHT-BY,INSTRUCTOR).* By switching the source and target entities and using relationships that are "inverses" of each other, we obtain two different facts that represent the same information. Assuming that information on inverse relationships is available, given one representation, the other could be inferred. Inverse information is simply a relationship between entities and, therefore, can be expressed with facts. To express inverse relationships, a special entity $\leftrightarrow$ is used. Facts in which $\leftrightarrow$ is the relationship are called *inversion facts*; their sources and targets are *inverse entities. (TEACHES,↔,TAUGHT-BY)* is an inversion fact to express that *TEACHES* and *TAUGHT-BY* are inverse entities. Given *(INSTRUCTOR,TEACHES, COURSE)* and *(TEACHES,↔,TAUGHT-BY)* the fact *(COURSE, TAUGHT-BY,INSTRUCTOR)* may now be inferred. Formally, inference by inversion is defined as follows:

(s,r,t) and (r,↔,r') ⇒ (t,r',s). (4)

Assuming that $\leftrightarrow$ is its own inverse (i.e. the database always includes the fact *(↔,↔,↔)*), inversion facts are guaranteed to come in pairs. For example, the inversion fact *(TAUGHT-BY,↔,TEACHES)* is readily inferred from the previous inversion fact. Consequently, given the facts *(COURSE,TAUGHT-BY,INSTRUCTOR)* and *(TEACHES ↔,TAUGHT-BY)*, the fact *(INSTRUCTOR,TEACHES, COURSE)* may be inferred. Formally, the fact *(↔,↔,↔)* guarantees that

(r,↔,r') ⇔ (r',↔,r).

3.5. Contradiction Facts

To increase 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 any two entities. Such relationships may be mathematical (i.e. $=$ and $>$) or linguistic (i.e. *LOVES* and *HATES*). To express contradiction relationships, a special entity $\perp$ is used, and such facts are called *contradiction facts.* Examples are *(=,⊥,>)* and *(LOVES,⊥,HATES).* As was the case with the entity $\leftrightarrow$, we assume that $\perp$ is its

[1] Notice that we assume that the semantics of facts such as *(EMPLOYEE,WORKS-FOR,DEPARTMENT)* is "*every* employee works for *at least one* department (although complete information on the particular departments for which every employee works may be unavailable)".

own inverse. That is, the fact $(\perp,\leftrightarrow,\perp)$ is part of the database. This ensures the symmetry of contradiction facts:

$(r,\perp,r') \Leftrightarrow (r',\perp,r)$.

3.6. Mathematical Facts

Often, a query may assert a mathematical relationship that the data must maintain. For example, "list all employees who earn more than 20000" or "list all students with average grade under 2.6". 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, *(25000,>,20000)* and *(2.2,<,2.6)*. While in this paper we do not address the issues of storage strategies for loosely structured databases, it is obvious that we may assume the existence of all relevant mathematical relationships, without actually storing them as ordinary facts.

In particular we include in the universe of entities all the numbers, as well as the special entities $<$ and $>$. For every two different number entities *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,≠,E2)*, depending on whether *E1* and *E2* are identical or not. Other mathematical relationships, such as $\leq$ and $\geq$ may be defined through simple inference rules.

As an example, consider the previous query about employees who earn over 20000. Its formal specification is:

$$Q(x) = (\exists y)((x,\in,EMPLOYEE) \wedge (x,EARNS,y) \wedge (y,>,20000)).$$

Assuming that the database includes the facts *(JOHN,∈,EMPLOYEE)*, *(JOHN,EARNS,25000)* and *(25000,>,20000)*, the answer to Q will include *JOHN*.

3.7. Inference by 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, *(TOM,ENROLLED-IN,CS100)* and *(CS100,TAUGHT-BY,HARRY)* imply a relationship between the entities *TOM* 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 *(TOM,ENROLLED-IN,CS100)* and the fact *(CS100,TAUGHT-BY,HARRY)* is the fact *(TOM,ENROLLED-IN.CS100.TAUGHT-BY,HARRY)*. Inference by composition is defined as follows:

$$(s,r,t),\ (t,u,v) \text{ and } (v,\neq,s) \Rightarrow (s,r.t.u,v). \quad (5)$$

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 ("paths") that relate John and Mary; e.g. *HUSBAND-OF*, *FATHER-OF.NANCY.DAUGHTER-OF* and *WORKS-FOR.PETER.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,LOVES,MARY)* and *(MARY,LOVES,JOHN)*, an infinite number of different composition facts would be generated; *(JOHN,x,MARY)* would then match an infinite number of different relationships.

4. Browsing

Imagine a customer entering a department store searching for a particular item. The most efficient method to locate this item is to consult a directory (or a knowledgeable employee) and proceed directly to the correct shelf. However, it may be the case that the customer cannot describe (or does not know) what is the item he is looking for; or, even if he does, perhaps the store is not organized in any meaningful way; or, even if he knows what he wants and the store is organized, a directory (or any other type of help) may be unavailable. In all these cases the customer must apply a different search technique, usually called *browsing*. Often, browsing is done by strolling along the aisles, adjusting direction and speed according to the items encountered and their "proximity" to the desired item. It may also involve "hit-and-miss" attempts, where the customer goes directly to a particular shelf, where he hopes the desired item will be found. While there is no doubt that the directory assisted approach is more efficient when applicable, in such situations as described above browsing is the only solution possible.

Similarly, most database architectures are constructed for efficient, schema based querying. If, however, the database is not organized, or its organization is not available to the user, or the user does not know exactly what to look for, this type of querying is rather futile. In our loosely structured database architecture, browsing is available in two different styles.

Navigation is the basic browsing tool for users who either do not know what to look for, or do not know enough about the database to formulate a standard query.

Navigation is analogous to strolling along the aisles of a store. On the other hand, a database user who attempts queries without sufficient familiarity with the database is like the store browser who makes a hit-and-miss attempt by going directly to a particular shelf in the store. This style of browsing will be called *probing*. What characterizes probing is that it will fail frequently. To assist such browsers we provide a mechanism for *automatic retraction*. In the remainder of this section we describe browsing by navigation. Probing is described in the next section.

4.1. Navigation

The process of navigation is based on template retrieval. These primitive queries allow the user to examine the neighborhood of a particular entity, pick an entity in that neighborhood, retrieve its own neighborhood, and so on. This interactive process continues until the user finds the data he needs.

Normally, the user supplies templates which have either one or two free variables. The answer is then represented as a single column (if the template had only one free variable), or in a two-dimensional table (if the template had two free variables).

In navigation queries we shall use the special symbol * in place of all independent variable names. For example, the template *(*,E,*)* matches *all* facts whose relationship is *E* (not only those whose source and target are identical), and is identical to the template *(x,E,y)*.

The following example demonstrates a simple navigation process. First, the template query *(JOHN,*,*)* is attempted with this result:

JOHN**			
∈	LIKES	WORKS-FOR	FAVORITE
PERSON	CAT	DEPARTMENT	-MUSIC
EMPLOYEE	FELIX	SHIPPING	PC#9-WAM
PET-OWNER	HEATHCLIFF	BOSS	PC#2-PIT
MUSIC-LOVER	MOZART	PETER	S#5-LVB
	MARY		

Next, the user may attempt *(PC#9-WAM,*,*)*, obtaining:

PC#9-WAM**			
∈	COMPOSED-BY	PERFORMED	FAVORITE
CONCERTO	MOZART	-BY	-OF
CLASSICAL		SIRKIN	JOHN
COMPOSITION		BARENBOIM	LEOPOLD

Finally, entering *(LEOPOLD,*,MOZART)*:

LEOPOLD*MOZART		
FAVORITE-MUSIC.	PC#9-WAM	.COMPOSED-BY
FATHER-OF		

The last example shows the power of composition as a browsing tool. The user may enter any two source and target entities, to obtain all the different associations between them.

While navigation provides a method for browsing, it is not different in principle from "normal" querying using the standard query language (template queries are a subset of the query language). Thus, navigation and querying may be interleaved: a user may submit a complex query, and use the answer as a starting point for browsing. For example, a complex query may be used to retrieve the name of the person which satisfies a particular predicate; this may then followed by browsing, in an attempt to uncover some interesting information about this person.

5. Probing

As noted earlier, probing is characterized by frequent failures: based on limited familiarity, the user may try a hit-and-miss strategy, which will often fail. The failure of a query provides no information as to the cause for failure. Consider the simple query "List all quarterbacks who have graduated from USC":

$$\mathcal{Q}(x) = (x,\in,QUARTERBACK) \wedge (x,GRADUATE\text{-}OF,USC).$$

If it fails, it may be simply because no quarterbacks have ever graduated from USC, but possibly because no *athletes* have ever graduated from USC, or because no quarterbacks have ever *attended* USC, or because particular positions of football players are not recorded in the database, or because USC is not a recognized database entity, etc.

In most database systems, a user who attempts such a query gets no little assistance when it fails. Unless the failure is due to erroneous syntax, the system will treat most failures simply as legitimate "empty answers". After such failures, a user with limited familiarity with the database, will have made very little progress towards his goal.

5.1. Broadness

Consider now this second query:

$$\mathcal{Q}'(x) = (x,\in,QUARTERBACK) \wedge (x,ATTENDED,USC).$$

$\mathcal{Q}'$ is identical to $\mathcal{Q}$, except for the substitution of the relationship *GRADUATE-OF* with the more general entity *ATTENDED*. Recall that by inference rule (1) the existence of the generalization fact *(GRADUATE-OF,≺, ATTENDED)* guarantees that if *(x,GRADUATE-OF, USC)* is a fact, then *(x,ATTENDED,USC)* is also a fact. Consequently, if $\mathcal{Q}$ succeeds, then $\mathcal{Q}'$ succeeds also. That is, the predicate $\mathcal{Q}$ *implies* the predicate $\mathcal{Q}'$, and the answer $\{\mathcal{Q}\}$ is *contained* in the answer $\{\mathcal{Q}'\}$.

The inference rules specified in (1) provide a method for

obtaining more general queries from a given query. Let $\mathcal{Q}$ be a query, let $(E,\prec,E')$ be a generalization fact, and assume that E appears in $\mathcal{Q}$. Let $\mathcal{Q}'$ be the query that is obtained by replacing a particular occurrence of E in $\mathcal{Q}$ with E'. Then $\mathcal{Q} \Rightarrow \mathcal{Q}'$. Given two queries $\mathcal{Q}$ and $\mathcal{Q}'$, if $\mathcal{Q} \Rightarrow \mathcal{Q}'$ then $\mathcal{Q}'$ is said to be *broader* than $\mathcal{Q}$. Clearly, if a query succeeds, all broader queries will succeed too, and if it fails, all narrower queries will fail.

An entity E' is a *minimal* generalization of E, if $(E,\prec,E')$ and $(E,\not\preceq,E')$ and there is no third entity X such that $(X,\not\preceq,E)$, $(X,\not\preceq,E')$, $(E,\prec,X)$ and $(X,\prec,E')$. Notice that an entity may have several minimal generalizations. A query $\mathcal{Q}'$ is said to be *minimally* broader than $\mathcal{Q}$, if it is obtained from $\mathcal{Q}$ by a *single* application of one of the inference rules specified in (1), using a minimal generalization.

For example, consider the following template query to list everybody who loves opera:

$\mathcal{Q}(x) = (x,LOVES,OPERA)$.

Its minimally broader queries include:

$\mathcal{Q}_1(x) = (x,ENJOYS,OPERA)$,

$\mathcal{Q}_2(x) = (x,LOVES,MUSIC)$,

$\mathcal{Q}_3(x) = (x,LOVES,THEATER)$.

The first query retrieves everybody who merely *enjoys* opera, and was obtained with the minimal generalization $(LOVES,\prec,ENJOYS)$. The other two queries retrieve, respectively, everybody who loves music, and everybody who loves theater. They were obtained with the minimal generalizations $(OPERA,\prec,MUSIC)$ and $(OPERA,\prec,THEATER)$.

Finally, given a query $\mathcal{Q}$, the set of all queries that are minimally broader than $\mathcal{Q}$ is called the *retraction set* of $\mathcal{Q}$.

5.2. Automatic Retraction

Every query may be regarded as a request to the database to "zoom in" on particular data. The failure of a query can then be attributed to "overzooming", that narrowed down the data too much. A "zoom out" may then be performed in an attempt to correct this. Such a "zoom-out" effect is achieved with retraction queries. Overzooming may have been caused by a strong relationship (a weaker one will be attempted), an entity which is too general (a more specific entity will be attempted), and so on. In each case the zoom out effect is achieved by relaxing the conditions that the data should satisfy, thus covering more "area" of data.

When a query fails we automatically attempt its retraction set. Every success of a query from this set is reported to the user with an indication of the generalization performed.

As an example, consider this query to retrieve the free things that all students love:

$\mathcal{Q}(x) = (STUDENT,LOVE,x) \wedge (x,COSTS,FREE)$.

Assuming that $(FRESHMAN,\prec,STUDENT)$, $(LOVE,\prec,LIKE)$, $(COSTS,\prec,\Delta)$ and $(FREE,\prec,CHEAP)$ are minimal generalizations, the following queries are minimally broader than $\mathcal{Q}$:

$\mathcal{Q}_1(x) = (FRESHMAN,LOVE,x) \wedge (x,COSTS,FREE)$,

$\mathcal{Q}_2(x) = (STUDENT,LIKE,x) \wedge (x,COSTS,FREE)$,

$\mathcal{Q}_3(x) = (STUDENT,LOVE,x) \wedge (x,COSTS,CHEAP)$,

$\mathcal{Q}_4(x) = (STUDENT,LOVE,x) \wedge (x,\Delta,FREE)$.

These queries retrieve, respectively, the free things that all *freshmen* love, the free things that all students *like*, the *cheap* things that all students love, and the things that all students love which are related to *FREE*.

When $\mathcal{Q}$ fails, the system may respond with the following menu:

```
Query failed. Retrying...
1. success with FRESHMAN instead of STUDENT
2. success with CHEAP instead of FREE
You may select...
```

The user may then select to display the results of any of these more general queries.

Notice that when a query fails and all retraction queries succeed, a "critical" point has been isolated, where every more general query is answerable, but the "conjunction" of these queries is unsuccessful. Such *critical failures* provide us with insight into the nature of the original failure. They point out where exactly the database is unable to satisfy our query. When a query fails and all its retraction queries fail also, it is obvious that the failure of the original query was insignificant. We then repeat the same process on each failed retraction query. The queries in this second wave are up one more level in the broadness hierarchy. This process continues, until some retrieval is successful (or it is abandoned by the user).

As more generalizations are attempted, more database entities are replaced with Δ or ∇. Eventually, there may be templates composed entirely of variables and Δ and ∇. Such templates represent weak restrictions, which frequently are meaningless. The generalization of queries with such templates is achieved by *deleting* them altogether.

Assume now that a query with a misspelling is attempted; for example $(JOHN,LOEVS,x)$. As *LOEVS* is not a database entity, it will never be replaced. After several generalizations in the source position, the query $(\Delta,LOEVS,x)$ will be generated, and fail. The failure of queries that do not have any broader queries is reported as "no such database entities".

6. Further Possibilities

The architecture we have outlined in this paper may be explored further to facilitate the use of the system. One possible extension is to provide a definition facility to implement new retrieval operators, based on the standard query language. Other operators may be defined to control the behavior of the inference system, or to allow update of the facts and the rules. Some of these possibilities are demonstrated below.

6.1. Operators

A navigation process, as simple as it is, requires that the user supplies the first template. This template must include at least one database entity in its proper position. A simple aid allows the user to obtain such start-up information: the operator ***try(e)*** takes an entity and returns all database facts that include this entity. This operator is implemented with the standard query $(x,y,z) \wedge ((x,=,e) \vee (y,=,e) \vee (z,=,e))$. With a couple of ***try***s even users completely unfamiliar with the database should be able to pick a starting point for navigation.

A database is a dynamic set of facts, that may be modified continuously by insertions, deletions and updates. As inference rules are representations of additional facts, they too may be edited dynamically. This allows us to turn inference rules off and on, at will. For example, if inference by composition is undesirable because it is too powerful (and expensive) it may be switched on to augment the database only before a particular retrieval, and switched off afterwards to reduce it again. The operators ***include(rule)*** and ***exclude(rule)*** achieve this effect.

When inference by composition is in effect, the database is augmented with *all* possible composition facts. This may have serious effect on the cost of query processing. To reduce this effect we may wish to allow only *limited* composition, by imposing an upper bound on the number of composition operations allowed in the inference of composition facts. The operator ***limit(n)*** sets the limit on the length of composition chains to be n. For example, $n=1$ disables composition altogether; $n=2$ allows composition, but facts that were obtained through composition cannot participate in further compositions; $n=\infty$ permits unlimited composition. It is usually the case that as the chain of compositions gets longer, the relationship between its two end entities becomes less significant (the length of such a path is sometimes called the *semantic distance* between these entities). Thus, control over the length of composition chains enables the user to determine at what "distance" a relationship ceases to be significant, and allows him to exclude from the database facts that involve insignificant relationships.

Representation of information as an unstructured heap of facts suitable for browsing, should not prevent structured views of this information. On the contrary, using the standard query language, the user may view this information as if it is structured according to different data models, such as the relational or the functional. For example, structuring the information in relations is possible with the retrieval operator ***relation***$(s,r_1{:}t_1,\ldots,r_n{:}t_n)$. This operator is implemented with the query $(y,\in,s) \wedge (x_1,\in,t_1) \wedge \ldots \wedge (x_n,\in,t_n) \wedge (y,r_1,x_1) \wedge \ldots \wedge (y,r_n,x_n)$. It returns a tabulated relation with $n+1$ columns. In each row of this table, the entity in the first position is an instance of s, and the entity in the i'th position ($i=2,\ldots,n$) is an instance of t_i. The entity in the first position is related to the entity in the i'th position ($i=2,\ldots,n$) through the relationship r_i. For example, ***relation***$(EMPLOYEE,WORKS\text{-}FOR{:}DEPARTMENT,EARNS{:}SALARY)$ returns

EMPLOYEE	WORKS-FOR:DEPARTMENT	EARNS:SALARY
JOHN	SHIPPING	$25000
TOM	ACCOUNTING	$27000
MARY	RECEIVING	$25000

Such relations are not necessarily in first normal form; that is, except for the first column, positions in this table may hold any number of entities.

6.2. Conclusion

The database architecture we have presented in this paper may be summarized by the following features: (1) It promotes databases which are only loosely structured: a heap of facts instead of highly structured data. (2) It erases the traditional dichotomy between "schema" and "data": all information is treated uniformly. (3) It incorporates a single mechanism for defining both inference rules and integrity constraints. (4) It supports browsing in two different styles: navigation and probing. (5) It has a sound formal basis, in the form of predicate logic, including a complete retrieval language.

This architecture is particularly suitable for modelling environments which are subject to constant evolution (or of which our perception is continuously evolving). Environments which do not lend themselves to "massive" classifications are captured more naturally by this architecture. In general, this alternative architecture can be applied whenever we prefer to trade retrieval efficiency, for minimal investment in organization.

Several important issues still remain to be investigated. Among them are suitable storage strategies, performance, and update of data. A planned implementation effort is expected to provide additional insight into these issues, as well as on the effectiveness of the browsing mechanisms.

Acknowledgements

I am grateful to my colleagues Dave Jefferson and Rick Hull for their important comments.

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.
The 3 Dimensional Information Space (3DIS).
Technical Report TR-84-303, University of Southern California, April, 1984.

[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 . North-Holland, 1976.

[4] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, 1981.

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

[6] M. Hammer and D. McLeod.
Database Description with SDM: A Semantic Database Model.
ACM Transactions on Database Systems 6(3):351-386, September, 1981.

[7] C. Herot.
Spatial Management of Data.
ACM Transactions on Database Systems 5(4):493-513, December, 1980.

[8] 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 Languages*, chapter 1. Springer-Verlag, to appear.

[9] J. Mylopoulos and H.K.T. Wong.
Some Features of the TAXIS Data Model.
In *Proccedings of the Sixth International Conference on Very Large Data Bases*, pages 399-410. Montreal, Canada, 1980.

[10] R. Reiter.
Towards a Logical Reconstruction of Relational Database Theory.
In M.L. Brodie, J. Mylopoulos and J.W. Schmidt (editors), *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, chapter 8. Springer-Verlag, to appear.

[11] M. Stonebraker and J. Kalash.
TIMBER: A Sophisticated Database Browser.
In *Proccedings of the Eighth International Conference on Very Large Data Bases*, pages 1-10. Mexico City, Mexico, 1982.

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