Constructing Queries from Tokens

Amihai Motro

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

Abstract

A database token is a value of either the data or the metadata Usually, such tokens are combined with formal language constructs to form queries In this paper we show how a given set of tokens may be completed to a proper query This process provides a useful means of communication between naive users and databases, allowing them to express simple requests by listing several tokens As the inferred query is always shown to the user, this process has a side effect of instructing the user in the proper use of the query language The method is described and demonstrated with relational databases, but its principles may be implemented with other databases as well

1. Introduction

Most query languages require their users to have specific retrieval goals, and then express them in a formal way This also implies that users must be familiar with the organization of each database that they access (which, in turn, requires adequate understanding of the data model used by the system)

Often, users may lack some of these prerequisites For example, a user may be aware that a database is available on students, courses and enrollments, and would like to find out the Grade Point Average (GPA) of a student called Smith However, due to insufficient experience with either the query language or the data model or this database, this user may be unable to utter more than "SMITH GPA", where he should be entering something like

retrieve GPA **of** STUDENT **where**
NAME=SMITH

or

$$Q(y) = \text{STUDENT}(x) \wedge \text{NAME}(x,\text{SMITH}) \wedge \text{GPA}(x,y)$$

The problem with SMITH GPA is, of course, that it is not a proper query But, then, this may be regarded as a limitation of the query interface used We could imagine an interface that "understands" such utterances by interpreting them in a unique way, i e complete them to proper queries If answers to such fragmented queries are always accompanied by the inferred queries, then users get a chance to reject interpretations which are incorrect (if the user is not sure whether an interpretation is correct, he may have to try the query and examine the data it retrieves) Displaying the interpretations also has instructional benefits, as it shows the user how the request should have been phrased

Consider the situation of a person who goes into a shop in a foreign country As he does not know the language, he utters whatever words he knows, in an attempt to describe what he needs The shopkeeper, eager to help, tries to interpret this request and fetches something (perhaps more than one thing) If it doesn't match his intentions, the customer will try to modify his request, perhaps by adding another word (if he observes that it is needed to disambiguate his previous request), or by eliminating a previous word (if he can identify that it led to the erroneous interpretation) The process may continue, until finally the interpretation is close enough to the intentions of the customer Of course, there is always the possibility that the customer walks away frustrated (or worse, that in the end he buys the wrong thing!), but often, the process will be carried out successfully (and the customer even picks up a new phrase in this language)

Almost every aspect of this situation has its direct analogy in database access (including the advantage of shopkeepers that speak one's language) Indeed, the interpretation of query utterances may be regarded as a case of *understanding unparsable input*, an active area of natural language research (for example, see [3, 5, 11]) Our approach here is to create a useful interface which is both cooperative and instructional (objectives often associated with "intelligence"), while using only the database at hand

We define *database tokens* to be the basic utterances, and then develop methods by which a set of tokens may be completed by the system to a proper query, which is then displayed to the user for his approval (as well as enlightment) The proper query is then submitted to the standard query processor Our *token interpreter* can therefore be regarded as an interface for naive users that should be used in tandem with a standard interface (Figure 1-1)

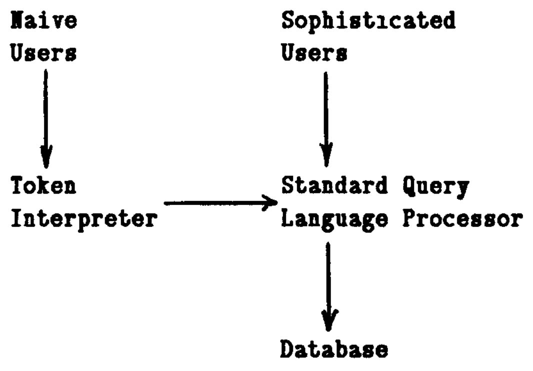

Figure 1-1: The Token Interpreter

Our work here is done in the context of the relational database model The primary reason for this is the widespread use of relational databases The principles could also be implemented with other data models, such as the network, the hierarchical or the functional data models

Our work recalls several efforts on the automatic connection of database relations [1, 8] Another related effort is System/U [4], the relational DBMS which is based on the universal relation assumption The main advantage of System/U is that it relieves the user from the responsibility of navigating within the relations, relying instead on the predefinition of schematic constructs called maximal objects Also related is work based on the universal relation schema assumption [6], which guarantees that a set of attributes uniquely determines a semantic connection between them

Similarly, we too are concerned with automatic inference of the connections required to answer a query Here, however, the emphasis is on an interface which is extremely simple In particular, it is free of any data model details the user who supplies the tokens need not know about concepts such as relations, schemas or attributes He need not even distinguish between *data values* and *metadata values* (i e relation names, attribute names, domain names) He only needs to supply a list of *words* We see the uniform treatment of data and metadata as an important convenience virtually all database interfaces perpetuate the dichotomy between data and metadata, when this distinction may be of concern to database designers, but is not always clear to database users In addition, we do not subscribe to any assumptions on the structure of the database, and our methods are intended to be used with any relational DBMS At times our methods require interaction with the user, such as to disambiguate a token, or to select between alternative interpretations (System/U would retrieve the union of the alternatives) Admittedly, as the "language of tokens" has no formal structures, it is considerably less expressive than regular query languages However, we must consider this as the price of doing away with all formalities

Our methods require a schematic representation of relational databases called dependency graphs This representation is described in Section 2, and Section 3 then defines queries and tokens The mechanism for interpreting tokens into queries is described and demonstrated in Section 4, and Section 5 shows how to handle ambiguous sets of tokens Section 6 describes a few useful extensions of the basic mechanism, and discusses complexity and implementation We conclude in Section 7 with a brief discussion of the results

2. Dependency Graphs

We assume relational schemas provide the following information There is a set of distinctly named *relations* With each relation there is an associated set of distinctly named *attributes*, one of which is designated as *key attribute*[1] Each attribute has an associated *domain*

As an example, Figure 2-1 defines a database UNIVERSITY with three relations STUDENT, COURSE and ENROLLMENT Each relation definition shows the attributes (the key attribute is underlined) and their associated domains Thus, the attribute NAME in relation STUDENT and the attribute STUDENT in relation ENROLLMENT are both of domain PERSON_NAME A small instance of this database is shown in Figure 2-2

STUDENT		
	<u>NAME</u>	PERSON_NAME
	MAJOR	ACADEMIC_DISCIPLINE
	GPA	NUMBER
COURSE		
	<u>C-NO</u>	COURSE_NUMBER
	DEPARTMENT	ACADEMIC_DISCIPLINE
	UNITS	NUMBER
ENROLLMENT		
	<u>E-NO</u>	ENROLLMENT_NUMBER
	STUDENT	PERSON_NAME
	COURSE	COURSE_NUMBER
	GRADE	LETTER_GRADE

Figure 2-1: Schema of Database UNIVERSITY

[1]For simplicity, we assume that all relations have simple (one attribute) keys Composite keys will be discussed later

STUDENT

NAME	MAJOR	GPA
Brown	Math	2 6
Chen	ElecEng	3 2
Klein	CompSci	2 8
Smith	Math	3 4

COURSE

C-NO	DEPARTMENT	UNITS
CS101	CompSci	4
CS202	CompSci	3
MATH270	Math	4
MATH370	Math	3
BIO425	Biology	4

ENROLLMENT

E-NO	STUDENT	COURSE	GRADE
E762	Smith	MATH370	C+
E824	Smith	CS101	A-
E628	Brown	BIO425	B+
E742	Brown	MATH370	B
E844	Klein	CS101	A
E722	Klein	MATH270	B-
E535	Chen	CS202	B

Figure 2-2: Example of Database UNIVERSITY

Each relational schema may be represented as a *dependency graph* The dependency graph has a node for each attribute and for each domain From each domain node there are directed arcs to all the nodes of the attributes that draw their values from this domain Such arcs are called *similarity* arcs From each node that corresponds to a key attribute there are directed arcs to all the nodes of the non-key attributes of the same relation Such arcs are called *dependency* arcs Similarity arcs connect attributes that use values of the same domain in the relational database these two attributes could serve as the basis for a natural join between their respective relations Dependency arcs represent *functional dependencies* each database value in the tail attribute determines a unique database value in the head attribute The dependency graph for database UNIVERSITY is shown in Figure 2-3 The dashed arcs are for similarities, the solid arcs are for dependencies Note that dependency graphs are not necessarily connected

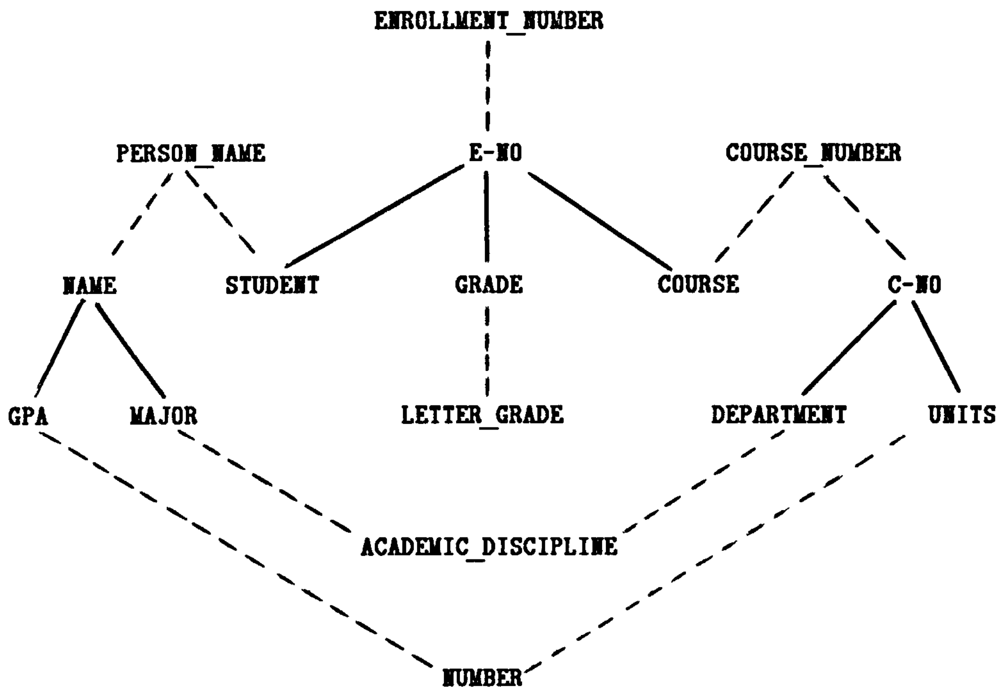

Figure 2-3: Dependency Graph for Database UNIVERSITY

When relational queries mention particular data values, they are always associated with particular metadata values For example, the query "**retrieve** STUDENT NAME **where** STUDENT MAJOR=MATH" implies that MATH is from the domain of attribute MAJOR By itself, the value MATH does not suggest any particular domain In this paper we assume the database management system stores domain information in an auxiliary relation called LEXICON with two attributes, VALUE and DOMAIN A tuple (V,D) in this relation indicates that value V is from domain D (and possibly appears in the database under some attribute of this domain) Some examples of LEXICON tuples are (BROWN, PERSON_NAME), and (B+,LETTER_GRADE) Given an arbitrary value, the system can use this LEXICON to find out its possible domains (and thus gain some understanding of its meaning) This relation is further discussed, along with other implementation issues, in Section 6

3. Queries and Tokens

Relational databases may be queried on the basis of their dependency graphs, using a formal *dependency language* which is based on predicate calculus

For each dependency arc we introduce a binary *predicate* named with its two participating attributes (tail attribute first) For example, NAME→MAJOR, C-NO→DEPARTMENT and E-NO→STUDENT The instantiation of predicate A→B for data values U and V (denoted U A→B V) is *true*, if and only if the relation from which this dependency was extracted has a tuple with values U and V for attributes A and B, respectively Thus, CHEN NAME→MAJOR ELECENG is *true*, but CS101 C-NO→ DEPARTMENT MATH is *false* In addition to these dependency predicates we assume there are built-in binary predicates such as $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$

With *variables*, predicates may be used to specify sets of data values For example, if x is a variable, then the predicate x STUDENT→MAJOR MATH specifies the set of students who are Math majors Using conjunction, disjunction and negation operations, predicates may be combined into *formulas* All variables in a formula are assumed to be existential quantifiers, unless they are declared to be free A formula which declares its free variables is called a *query* Let $Q(x_1, \ ,x_n)$ be a query with free variables $x_1, \ ,x_n$ The *value* of Q is the set of all data values $v_1, \ ,v_n$ which satisfy the formula [2]

As an example, assume we want to retrieve the name and major of all students with a GPA over 3 0 and who received either A or B in CS101 We assign variables x, y, z and w to range over PERSON_NAME, ACADEMIC_DISCIPLINE, NUMBER and ENROLLMENT_NUMBER, respectively The statement "**retrieve** (x,y)" declares x and y to be free, it is followed by a formula with seven predicates

```
let x be from domain PERSON_NAME
let y be from domain
  ACADEMIC_DISCIPLINE
let z be from domain NUMBER
let w be from domain
  ENROLLMENT_NUMBER
retrieve (x,y) where
  (x NAME→MAJOR y) ∧
  (x NAME→GPA z) ∧
  (z > 3 0) ∧
  (w E-NO→STUDENT x) ∧
  (w E-NO→COURSE CS101) ∧
  ((w E-NO→GRADE A) ∨
  (w E-NO→GRADE B))
```

Like most other query languages, this dependency language involves complex structures and rigid syntax In contrast, tokens are very elementary a *database token* is either a data value or a metadata value (i e a relation name, an attribute name or a domain name) Some examples are

ENROLLMENT, STUDENT, MAJOR, SMITH and CS101

[2]Note that negation is well defined a negated predicate is *true*, if the relevant relation does not have a tuple with these values As existential quantification is always done globally, our formulas cannot simulate universal quantification If logical completeness is important, universal quantifiers should be added

Each set of such tokens is considered as an attempt to formulate a query Two examples of token sets are

SMITH, GPA

and

STUDENT, MAJOR, BIO425, F

Our techniques for interpretation of database tokens will result in queries to retrieve, respectively, the GPA of Smith, and the name and major of the students who are enrolled in BIO425 and failed

4. Interpreting Sets of Tokens

Consider again the query to retrieve the name and major of all students with GPA over 3 0 Assume that for each of its six dependency predicates we *mark* the corresponding arc in the dependency graph (together with its end nodes) Also, assume that whenever two marked nodes are connected to the same domain (such as STUDENT and NAME) we mark the path between them (e g from each to PERSON_NAME) The result is a connected subgraph, which is shown in Figure 4-1

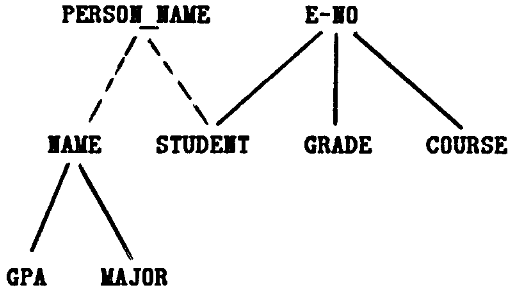

Figure 4-1: The Connected Subgraph for Our Example Query

While not all proper queries correspond to a single connected subgraph, a query that corresponds to several disjoint subgraphs could safely be regarded as a collection of independent queries that were submitted together Based on this observation, we may assume that only queries that correspond to a single connected subgraph are permitted Note that the mapping of queries into graphic representations is not injective and different queries may correspond to the same connected subgraph

With dependency graphs we can translate the problem of interpreting a set of database tokens into a graph problem Each token corresponds to a node in the graph, and this scattered set of nodes is a model of the request Its missing parts will be provided by the nodes and arcs necessary to connect them Thus, the interpretation we provide for a given set of database tokens is a query that corresponds to a subgraph that connects their corresponding nodes This process involves two issues (1) how to connect the nodes into a subgraph, and (2) which query corresponds to this subgraph These issues occupy much of the remainder of this paper

One constraint that we place on the connecting subgraph is that it should be *minimal*, i e there should be no strict subgraph of it that connects the given nodes Obviously, such subgraphs are always trees, and the problem is known as the *Steiner tree* problem [2] (This problem is a generalization of the *minimum spanning tree* problem) Still, this constraint does not guarantee uniqueness, as there may be several such trees, and each may serve as the basis for an interpretation Another issue, which will be addressed later, is the appropriate algorithm for finding minimum spanning trees and its complexity

We defer to a later section the discussion of how to obtain the desirable minimum spanning tree from the given nodes, and at this point assume that there is a procedure that performs this task We also assume that the given set of tokens does not include replications, and that all tokens are recognizable as values of either the data or the metadata We divide the process of interpreting a set of tokens into a proper query into six steps

1. Mark the dependency graph. For each token we *mark* a node in the dependency graph if the token is an attribute name, we mark the attribute node, if it is a domain name, we mark the domain node, if it is a relation name, we mark the node of its key attribute[3], otherwise, we assume it is a data value and we mark the node which corresponds to its domain (obtained from relation LEXICON) For example, consider the tokens STUDENT and MATH The first is an attribute name and we mark the node by this name The other is a data value and we mark the node ACADEMIC_DISCIPLINE

2. Obtain minimum spanning trees. Our next step is to obtain a minimum tree in the dependency graph that spans the marked nodes If a minimum spanning tree cannot be found (due to disconnectivity), then no query may be inferred from this set of tokens If several trees exist, then there should be a method for selecting one as the basis for interpretation As we stated earlier, we assume that a procedure that performs this task is available In our example, there are several minimum spanning trees One connects STUDENT and ACADEMIC_DISCIPLINE through PERSON_NAME, NAME and MAJOR, another connects them through E-NO, COURSE, COURSE_NUMBER, C-NO and DEPARTMENT These two trees correspond to different meanings of the token MATH (a major and a department) Additional trees exist that make connections through the node NUMBER For demonstration purposes we describe the rest of the process for the first two alternatives

3. Instantiate predicates. Next, we associate a variable with each node of the tree For each dependency arc we instantiate its predicate between the variables associated with its two attribute nodes For each similarity arc we instantiate the *equality* predicate between the variable associated with its domain node and the variable associated with its attribute node In our example, five variables x_1, ,x_5 are required for the first tree They are associated, respectively, with STUDENT, PERSON_NAME, NAME, MAJOR and ACADEMIC_DISCIPLINE Four predicates are instantiated

[3]Thus, a token which is a relation name is in effect mapped into the token which is the key attribute of that relation

$x_2 = x_1$
$x_2 = x_3$
x_3 NAME→MAJOR x_4
$x_5 = x_4$

The second tree requires seven variables y_1, ,y_7, which are associated, respectively, with STUDENT, E-NO, COURSE, COURSE_NUMBER, C-NO, DEPARTMENT and ACADEMIC_ DISCIPLINE The instantiated predicates are

y_2 E-NO→STUDENT y_1
y_2 E-NO→COURSE y_3
$y_4 = y_3$
$y_4 = y_5$
y_5 C-NO→DEPARTMENT y_6
$y_7 = y_6$

4. Substitute known data values. For each node that was marked because of a data value we now substitute the node variable with the value in all predicates in which it occurs In our example we substitute MATH for x_5 in the first alternative, and for y_7 in the second alternative We obtain

$x_2 = x_1$
$x_2 = x_3$
x_3 NAME→MAJOR x_4
MATH = x_4

and

y_2 E-NO→STUDENT y_1
y_2 E-NO→COURSE y_3
$y_4 = y_3$
$y_4 = y_5$
y_5 C-NO→DEPARTMENT y_6
MATH = y_6

5. Eliminate equality predicates. Next, we eliminate all the instantiations of the *equality* predicate one by one, after we perform the necessary substitutions in the other predicates In our example, we obtain

x_1 NAME→MAJOR MATH

and

y_2 E-NO→STUDENT y_1
y_2 E-NO→COURSE y_5
y_5 C-NO→DEPARTMENT MATH

6. Determine free variables and construct queries. Finally, we combine the predicates into a formula through multiple conjunctions In this formula, variables of nodes that were marked in the initial step (or variables of nodes connected by similarity arcs to nodes marked in the initial step) are considered free variables (in our example, x_1 and y_1) All other variables are considered existential (in our example, y_2 and y_5) The final queries that correspond to the tokens STUDENT, and MATH are as follows (we use notation which is somewhat less mathematical that that introduced in Section 3)

```
let x be from domain PERSON_NAME
retrieve (x) where
x is NAME of STUDENT having
    MAJOR MATH
```

and

```
let x be from domain PERSON_NAME
let y be from domain
    ENROLLMENT_NUMBER
let z be from domain
    COURSE_NUMBER
retrieve (x) where
y is E-NO of ENROLLMENT having
    STUDENT x and
y is E-NO of ENROLLMENT having
    COURSE z and
z is C-NO of COURSE having
    DEPARTMENT MATH
```

Thus, the two interpretations of the tokens STUDENT and MATH are "retrieve the names of all students who are Math majors" and "retrieve the names of all students who are enrolled in Math courses" The corresponding answers are

NAME
BROWN
SMITH

and

NAME
BROWN
KLEIN
SMITH

As another example, consider the tokens STUDENT, MATH and CS101 Again, there will be two interpretations, however, one interpretation will instantiate a predicate that will falsify the query

```
   MATH is DEPARTMENT of COURSE having
      C-NO CS101
```

The other alternative will retrieve all the Math majors enrolled in CS101

As a third example, the tokens STUDENT, MAJOR and CS101 will be interpreted as a query with two free variables, to retrieve the names and majors of all the students who are enrolled in CS101

```
   let x be from domain PERSON_NAME
   let y be from domain
      ENROLLMENT_NUMBER
   let z be from domain
      ACADEMIC_DISCIPLINE
   retrieve (x,z) where
   x is NAME of STUDENT having
      MAJOR z and
   y is E-NO of ENROLLMENT having
      STUDENT x and
   y is E-NO of ENROLLMENT having
      COURSE CS101
```

The answer is

NAME	MAJOR
KLEIN	COMPSCI
SMITH	MATH

Notice that our process handles single tokens satisfactorily If the token is a *metadata value*, the process will produce a query to retrieve all the values of a domain For example, the token COURSE results in

```
   let x be from domain
      COURSE_NUMBER
   retrieve (x)
```

which will list all course numbers A single token which is a *data value* generates no predicates, and therefore results in no operation (The user should be prompted for more information)

Sets of tokens that include no metadata values require no special handling either Because they contain database values only, they generate queries without any free variables These *propositions* are answered by TRUE or FALSE For example, the tokens KLEIN, CS101 and A are interpreted with the following proposition

```
   let x be from domain
      ENROLLMENT_NUMBER
   retrieve () where
   x is E-NO of ENROLLMENT having
      STUDENT KLEIN and
   x is E-NO of ENROLLMENT having
      COURSE CS101 and
   x is E-NO of ENROLLMENT having
      GRADE A
```

Its answer is

```
   TRUE
```

Sometimes, a set of tokens may include both a domain name and a database value from this domain (the domain node is then marked twice) This presents no problems, as the usual process will instantiate a variable for this node and then bind it to the particular data value The final effect will be as if the domain token was not included

5. Resolving Ambiguities

As already mentioned, a given set of tokens may be connected with several minimum spanning trees, which will lead to several different queries The given set of tokens is then called *ambiguous* We distinguish between three kinds of ambiguity

1 **Multiple trees for the same nodes.** The set of marked nodes (obtained from the given tokens) may be connected with more than one minimum spanning tree The example which demonstrated the process was of this kind

2 **Alternative sets of nodes.** A given data token may belong to more than one domain Together with the nodes marked by the remaining tokens, each of the possible domain nodes forms a valid set of marked nodes For example, BROWN may be known as both PERSON_NAME and COLOR, which, considered together with the attribute CAR, may lead to either "all cars owned by Brown", or "all brown cars"

3 **Multiple interpretations for the same tree.** Several values may be from the same domain, which is then marked several times

In such cases, several interpretations are possible even for the same tree For example, COURSE, BROWN and KLEIN could be interpreted as the courses in which *either* student is enrolled, or as the courses in which *both* are enrolled

There are three ways to approach ambiguities, and our methods here combine all three

1 **Determine correct interpretation.** The most desirable solution is, of course, a procedure which selects a single interpretation as the correct one Unfortunately, it is often impossible to justify any particular interpretation over the others A less ambitious (but more feasible) approach is to eliminate certain interpretations as incorrect, and perhaps rank the remaining interpretations for likelihood of being correct

2 **Interact with the user.** Alternatively, we may try to approach the user for further clarification This method is usually simpler and often cheaper, though sometimes the user may be unable to offer substantial assistance For example, the user may be asked to disambiguate a token ("is BROWN a COLOR or a PERSON_NAME?") Or the system may only request more tokens, in the hope that they will strengthen certain alternatives, and possibly invalidate others Or the user may be offered to select from several different interpretations of his token set

3 **Pursue multiple interpretations.** Finally, if the number of alternatives is relatively small, the system could pursue them all and present their outcome to the user for his decision

When the user enters a token, the system looks it up immediately in the lexicon If multiple entries are found for this token, then the system displays the different entries and requests the user to select among them Thus, ambiguities of the second kind are resolved instantly with user assistance

Ambiguities of the first kind are handled with a *prune-and-rank* procedure, which discards some interpretations and then ranks the remaining interpretations for likelihood of being correct The pruning phase is based on the following observation Whenever a domain node is used in the spanning tree, an assumption is made that the semantics of the attribute nodes which it connects are similar This is a risky assumption Sometimes the risk is low in our example, the attributes C-NO and COURSE have almost identical meanings However, GPA and UNITS have very little in common This suggests that interpretations that contain fewer such connections are more likely to be correct Consequently, given two interpretations we will prefer the one that makes the least number of such "joins" Thus, among all minimum spanning trees we prefer those that have the smallest number of similarity arcs, and discard the others Our ranking of the remaining acceptable trees is based on their *size* the smaller the tree, the more likely its interpretation is the correct one This measure is based on the observation that unlikely interpretations require more intermediate concepts to connect the set of tokens provided by the user, resulting in larger trees In the example that demonstrated the process, each of the two trees that were pursued has one such "join", but the first tree ("retrieve the names of all students who are Math majors") is smaller and is therefore ranked before the second ("retrieve the names of all students who are enrolled in Math courses") Several other trees are discarded altogether because they make more "joins" (e g "retrieve the names of all students whose GPA is identical to the units of some Math course")[4] After this procedure is performed, the different queries are presented to the user as possible interpretations of his tokens, and he is asked to select one If he declines, they are attempted sequentially, in their order of likelihood

[4]Some extensions of the relational model use the concept of *role*, to distinguish between attributes that are of the same domain, but still express different semantics As this feature eliminates the risk of "joins", our selection procedure under such a model will be based on size only

The last kind of ambiguity is the most difficult to resolve. Consider the tokens DEPARTMENT, BROWN and KLEIN (where the last two are both from domain PERSON_NAME) and the minimum spanning tree that connects them through STUDENT, E-NO, COURSE, COURSE_NUMBER and C-NO. The interpretation is to retrieve the departments which offer courses in which these students are enrolled. However, several variations are possible here, and they retrieve different sets of departments: those which enroll *either* BROWN or KLEIN; those which enroll *both;* and those which enroll both in the *same* course. To understand these alternatives, assume the tokens DEPARTMENT and BROWN and the tokens DEPARTMENT and KLEIN are considered separately. For each pair of tokens a chain of variables is instantiated over the path that connects them. In considering the three tokens together, we may choose to keep the chains separate, or we may identify some of the corresponding variables. The first interpretation is for separate chains; the second identifies the two DEPARTMENT variables; and the third also identifies the two COURSE variables. In such cases we adopt the safer approach of not identifying any of the variables. The effect is as if the user repeated the query twice: first with BROWN and then with KLEIN. It is also possible to show all the different interpretations to the user and have him select one.

6. Extensions and Implementation Considerations

Until now we have assumed that all relations have simple keys. However, no significant modifications are required to handle *composed keys*. In addition to the attribute nodes for every component of the key, the dependency graph should include an extra node for the composed key, with dependency arcs to each component node, as well as to the other attributes of the relation. Thus, if in the database UNIVERSITY, relation ENROLLMENT did not have the key attribute E-NO, but instead the composed key (STUDENT,COURSE), we would introduce a new node for the composed key, and the dependency graph would be exactly like the one obtained before. This extra node corresponds to the structured attribute that many database management systems introduce to handle composed keys (for example, [10]).

With simple extensions, the power of this naive query language can be increased. One example are *arithmetic comparisons*. Given the tokens NAME, GPA, $>$ and 3.0, the following query will be generated:

$$(x \text{ NAME}\rightarrow\text{GPA } y) \wedge (y > 3.0).$$

Another simple extension will allow users to refer to attribute names by example. A user who wants to know the numbers of math courses that give 3 units of credit, but does not know the token C-NO, will be able to supply instead an *example* of a course number he knows, e.g. CS101 (to indicate that CS101 is only an example, it should be prefixed by ~). A token set that will accomplish this request is: ~CS101, MATH, 3, UNITS. Note that the last token serves only as reinforcement; a proper interpretation will also be produced without it.

The query interpretation mechanism described here may be implemented with any relational database management system. The only modification needed is to add a lexicon feature (see below). As the dependency language queries produced by the interpreter are easy to translate into other query languages, this interface can be implemented to produce queries in a language already supported by the system. Note that the database itself is never modified, except for the additional relation LEXICON, and all other applications and user interfaces are unaffected by this mechanism.

The relation LEXICON must not be modified by users; the system should update it automatically, to reflect user updates to other relations. This is similar to the way that secondary indexes are handled in some relational systems (for example, [9]). A similar device was used by the relational interface BAROQUE, to provide a semantic network view of relational databases, and thus offer elaborate browsing functions [7]. The cost of this relation, in terms of the additional space to store this relation and the additional computation for its initialization and its continuous update, is comparable to the cost of a secondary index on every database attribute Except, of course, that there is no need to store numbers in the lexicon, as their domain is apparent, usually, numbers account for a substantial part of the database If the required storage is still prohibitive, it is possible to implement the lexicon only in part, by inverting on selected attributes only Thus, tokens of certain domains will not be recognized, and the system will have to extract the domain information by means of a dialogue

One difficulty that must be considered is that the algorithm for finding minimum spanning trees in a graph is known to be NP-Complete [2] However, its actual complexity can be controlled quite tightly by the following restrictions First, we may limit the number of tokens users are expected to provide, so that the number of marked nodes does not exceed 4 or 5 This restriction is quite safe, as the type of requests anticipated from users of this interface normally would not involve very many tokens (note that tokens which are from the same domain only mark a single node, so token sets can actually be much larger) Second, we may limit our search to spanning trees which require at most 2 or 3 unmarked nodes to connect any two marked nodes (i e sets of tokens which can only be connected by long paths of unmarked nodes are considered disconnected) Again, this is a reasonable assumption, as the significance of a possible association between two tokens decreases rapidly with the number of links necessary to connect them (note that in such cases, the process may be retried if the user provides an additional token to help associate the previous tokens) Finally, we may develop heuristic methods which construct spanning trees that are not necessarily minimal, but are compact enough to provide a reasonable basis for an interpretation

7. Conclusion

We have described a new kind of user-database interface which requires neither a formal language, nor any understanding of the organization of the database Our mechanism performs a task which may be called intelligent conclude a request from a set of tokens By returning its interpretations of token set requests, it also instructs users in the proper application of the formal language

Of course, the "language" of tokens is not complete in any sense a large number of requests cannot be phrased with tokens alone Still, tokens enable naive users to express a substantial number of popular queries in an extremely simple and informal way The combination of our token interpreter with a standard query interface, will result in a very effective interface

Our methods for interpreting a set of tokens sometimes require that the user be consulted, but most of the time we preferred to interpret the tokens provided by the user in some simple default way Another possible direction, which is currently under investigation, is to strengthen the interactive capabilities of the interface Thus, in effect the interface will start with the tokens, and then help the user construct his query Such a dialogue-based interface will remove some of the uncertainties that are part of our token interpretation methods Also, it will allow the user to specify a richer set of requests

References

[1] C R Carlson and R S Kaplan
A General Access Path Model and its Application to a Relational Database System
In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 143-154 1976

[2] S Even
Graph Algorithms
Computer Science Press, 1979, pages 24-26,225-226

[3] P J Hayes and J G Carbonell
Multi-Strategy Construction Specific Parsing for Flexible Data Base Query and Update
In *Proccedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 432-439 Vancouver, B C , 1981

[4] H F Korth et al
System/U A Database System Based on the Universal Relation Assumption
ACM Transactions on Database Systems 9(3) 331-347, September, 1984

[5] S Kwasny and N K Sondheimer
Ungrammaticability and Extra-Grammaticability in Natural Language Understanding Systems
In *Proceedings of the 17th Annual Meeting of the Association for Computational Linguistics* 1979

[6] D Maier
The Theory of Relational Databases
Computer Science Press, 1983, Chapter 12

[7] A Motro
BAROQUE A Browsing Interface to Relational Databases
Technical Report TR-8-312, Computer Science Department, University of Southern California, August, 1984
To appear in the ACM Transactions on Office Information Systems

[8] S L Osborne
Towards a Universal Relation Instance
In *Proceedings of the Fifth International Conference on Very Large Data Bases*, pages 52-60 Rio de Janeiro, Brazil, 1979

[9] M Stonebraker et al
The Design and Implementation of INGRES
ACM Transactions on Database Systems 1(3) 189-222, September, 1976

[10] *UNIFY Reference Manual*
3 0 edition, UNIFY Corporation, 1983

[11] R M Weischedel and J E Black
Responding Intelligently to Unparsable Inputs
American Journal of Computational Linguistics 6(2), 1980