FLEX: A Tolerant and Cooperative User Interface to Databases

AMIHAI MOTRO

Abstract—**FLEX is a user interface to relational databases that can be used satisfactorily by users with different levels of expertise. FLEX is based on a formal query language, but is *tolerant* of incorrect input. It never rejects queries; instead, it *adapts* flexibly and transparently to their level of correctness and well-formedness, providing interpretations of corresponding accuracy and specificity. The most prominent design feature of FLEX is the smooth concatenation of several independent mechanisms, each capable of handling input of decreasing level of correctness and well-formedness. Each input is "cascaded" through this series of mechanisms, until an interpretation is found. FLEX is also *cooperative*. It never delivers empty answers without explanation or assistance. By following up each failed query with a set of more general queries, FLEX determines whether an empty answer is *genuine* (it then suggests related queries that have nonempty answers), or whether it reflects *erroneous presuppositions* on behalf of the user (it then explains them).**

Index Terms—**Adaptivity, cooperation, databases, query languages, user interface, tolerance.**

I. Introduction

A COMMON method for accessing databases is via query language interfaces. A query language interface defines a formal language, in which all retrieval requests must be expressed. The main advantages of query language interfaces are their *generality* (the ability to express arbitrary requests) and their *unambiguity* (each statement has clear semantics). However, using query language interfaces requires considerable proficiency. Users must understand the principles of the underlying data model, they must have good knowledge of the query language, and they must be familiar with the contents and organization of the particular database being accessed. In the absence of even some of this prerequisite knowledge, using such interfaces can become very inefficient and frustrating. Hence, most query language interfaces do not accommodate naive users very well.

For such users, several alternative types of interfaces have been developed, including form and menu-based interfaces, graphical interfaces, natural and pseudonatural language interfaces, and browsers. These interfaces are oriented towards nonprogrammers, and therefore require only limited computer sophistication. Expressing requests may be as simple as selecting from a menu or a filling a form, and familiarity with the contents or organization of the database is usually not required. However, naive user interfaces usually achieve simplicity and convenience at the price of expressivity. Also, as users acquire more expertise, these interfaces tend to become more tedious to use.

Thus, it appears that no single user–database interface exists that can service satisfactorily both experts and naive users. Perhaps the only exception are the natural language interfaces. Ideally, such interfaces should be able to service satisfactorily all types of users. Unfortunately, existing natural language interfaces have two major problems: they require enormous investment to capture the knowledge that is necessary to understand user requests, and even the best systems are prone to errors.

This paper reports on research to develop a single interface, that may be used satisfactorily by users with different levels of expertise. This interface, called FLEX, is based on a formal query language, but is *tolerant* of incorrect input. It never rejects queries; instead, it adapts flexibly and transparently to their level of correctness, providing an interpretation at that level. Consequently, it can service a wider variety of users. FLEX is also *cooperative*. It never delivers empty answers without explanation or assistance. This tolerant and cooperative behavior is modeled after human behavior, and is thus reminiscent of natural language interfaces.

The design of FLEX is highly modular, consisting of various mechanisms for processing requests of different levels of well-formedness. Each user input is processed by several such mechanisms until an interpretation is obtained. Initially, the input is processed by a query *parser* to determine whether it constitutes a proper formal query. If parsing is successful, the query is executed. Otherwise, the input is processed by a query *corrector*, that attempts to salvage the query by applying various transformations. The corrector is usually successful whenever the input exhibits recognizable structures, and its interpretations are mostly safe. If the corrector fails to produce an interpretation, the input is processed by a query *synthesizer*, that attempts to conclude proper queries from words that are recognized in the input. As these interpretations are not entirely safe, they are offered as suggestions, and are subject to refinements by the user. Finally, if the synthesizer fails to produce an interpretation, a *browser* is engaged to

Manuscript received November 28, 1989; revised October 27, 1989. This work was supported in part by NSF Grant IRI-8609912 and by an Amoco Foundation Engineering Faculty Grant.

The author is with the Computer Science Department, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9035100.

display frames of information extracted from the database on the recognized input words. Hence, FLEX never rejects queries, and the accuracy and specificity of its interpretations correspond to the correctness and well-formedness of the input.

FLEX then observes the outcome of the final query: if the answer is empty, the original query is passed to a query *generalizer*, which issues a set of more general queries to determine whether the empty answer is *genuine* (it then suggests related queries that have nonempty answers), or whether it reflects *erroneous presuppositions* on behalf of the user (it then explains them).

Because it is engaged only *when* needed and only *as much* as needed, FLEX can be used satisfactorily by experts as well as novices. For example, input which is a perfect formal query will be executed immediately without any modification; while input which is a single word will flow through the entire sequence of mechanisms until finally it will result in a frame of information about this word.

FLEX was designed to work with relational databases. It was fully implemented as a front-end for the relational databases system INGRES [10]. A concise outline of the preliminary design of FLEX may be found in [21].

The remainder of this paper is organized as follows. Section II establishes preliminary concepts and definitions. The next four sections are devoted to the individual mechanisms of FLEX: Section III describes the query parser and the query corrector, Section IV describes the query synthesizer, Section V describes the browser, and Section VI describes the query generalizer. Section VII discusses the implementation, and Section VIII concludes with a brief summary.

II. Preliminaries

This section establishes concepts and definitions that are global to FLEX. It defines the data model and its formal language, it describes the knowledge used by the various mechanisms, and it presents an overview of the architecture of FLEX.

A. The Data Model

The following definition of relational databases is assumed. A database is a set of *relations*. For each relation there is a set of distinctly named *attributes*, some of which are designated as *key attributes*. Each attribute has an associated *domain*, and each domain has an associated *type*. From the information on the keys, the database system can infer existing *functional dependencies*: in each relation, every nonkey attribute is functionally dependent on the key attributes. From the information on the domains, the database system can infer the *allowable joins*: two relations may be joined if and only if they have a common domain for at least one of their attributes. Type information is used to allocate storage, to determine which operations are allowed with the elements of the domain, and to assist in query generalization. For simplicity, we consider only two types, STRING and NUMBER. The only parameter of the type STRING is its *length*. The type NUMBER has three parameters: *minimum*, *maximum* and *delta*; the first two specify the allowable range; the last one fixes the size of a "notch" in this range.

Names of relations, attributes, and domains must all be distinct (i.e., the same name cannot be used for a relation and an attribute, or a relation and a domain, or an attribute and a domain). However, attributes in different relations may have the same name, if they have the same domain.

Fig. 1 defines a database UNIVERSITY that will be used in the examples. The database has four relations: STUDENT, DEPARTMENT, COURSE, and ENROLLMENT. Each relation definition shows the attributes (key attributes are underlined) and their associated domains and types. Thus, the attribute MAJOR in relation STUDENT and the attribute D-NAME in relation COURSE are both of domain ACADEMIC_DISCIPLINE, which is of type STRING. A small instance of this database is shown in Fig. 2.

B. The Formal Language

The formal language of FLEX consists of the following statement, reminiscent of SQL's **select** statement [5]:

retrieve $attribute_1, \ldots, attribute_n$
from $relation_1, \ldots, relation_m$
where *condition*

condition is either a *primitive term* of the form *attribute* θ *value* or $attribute_1\ \theta\ attribute_2$ (where θ is a comparator such as $=, \neq, <, >, \leq, \geq$), or a combination of such terms with the logic connectors **and, or,** and **not**. The answer to this query is defined by a product of all the relations named in the **from** clause, followed by a selection according to the condition in the **where** clause, followed by a projection onto the attributes named in the **retrieve** clause. If two attributes in different relations are named identically, they are differentiated by including the relation name: *relation.attribute*. If more than one version of the relation is needed in the query, they are differentiated by an index: *relation*.1.*attribute*,*relation*.2.*attribute*, etc. If the **where** clause is omitted altogether, the selection condition is assumed to be *true*.

For example, to retrieve the names and majors of the students enrolled in courses offered by Computer Science Department, one issues the following query:

retrieve S-NAME, MAJOR
from STUDENT, ENROLLMENT, COURSE
where STUDENT.S-NAME=ENROLLMENT.S-NAME
and ENROLLMENT.C-NO=COURSE.C-NO
and COURSE.D-NAME="COMPSCI"

C. The Dictionary, the Lexicon, and The Thesaurus

A database consists of values (the *data*), which are organized according to a schematic definition (the *metadata*). Elements of the data and the metadata will be referred to collectively as database *tokens*. FLEX uses three special relations, called DICTIONARY, LEXICON, and THESAURUS to store information about data and metadata.

STUDENT			
	S-NAME	PERSON_NAME	STRING(20)
	MAJOR	ACADEMIC_DISCIPLINE	STRING(10)
	GPA	NUMBER_GRADE	NUMBER(0,1,0.2)
DEPARTMENT			
	D-NAME	ACADEMIC_DISCIPLINE	STRING(10)
	COLLEGE	COLLEGE_NAME	STRING(10)
	CHAIRPERSON	PERSON_NAME	STRING(20)
COURSE			
	C-NO	COURSE_NUMBER	STRING(8)
	D-NAME	ACADEMIC_DISCIPLINE	STRING(10)
	UNITS	UNIT_NUMBER	NUMBER(1,12,1)
ENROLLMENT			
	S-NAME	PERSON_NAME	STRING(20)
	C-NO	COURSE_NUMBER	STRING(8)
	GRADE	LETTER_GRADE	STRING(2)

Fig. 1. Schema of database UNIVERSITY.

STUDENT

S-NAME	MAJOR	GPA
BROWN	MATH	2.6
CHEN	ELECENG	3.4
KLEIN	COMPSCI	2.8
SMITH	MATH	3.2

DEPARTMENT

D-NAME	COLLEGE	CHAIRPERSON
BIOLOGY	SCIENCE	MILLER
COMPSCI	ENGINEER	DAVIS
ELECENG	ENGINEER	KING
MATH	SCIENCE	FOX

COURSE

C-NO	D-NAME	UNITS
CS101	COMPSCI	4
CS202	COMPSCI	3
MATH270	MATH	4
MATH370	MATH	3
BIO425	BIOLOGY	4

ENROLLMENT

E-NO	S-NAME	C-NO	GRADE
E762	SMITH	MATH370	C+
E824	SMITH	CS101	A−
E628	BROWN	BIO425	B+
E742	BROWN	MATH370	A−
E844	KLEIN	CS101	A
E722	KLEIN	MATH270	B−
E535	CHEN	CS202	B

Fig. 2. Instance of database UNIVERSITY.

The DICTIONARY relation stores the metadata in the following form: DICTIONARY=(RELATION,ATTRIBUTE, DOMAIN,TYPE,KEY). An example of a dictionary tuple is (STUDENT,S-NAME,PERSON_NAME,STRING(20),YES), which states that relation STUDENT has attribute S-NAME, which is of domain PERSON_NAME, has type STRING(20), and is part of the key.

The LEXICON relation is a mapping of data onto metadata: each data token is associated with the relation and attribute in which it appears (and, therefore, with its domain). Given an arbitrary data token, the system can use this lexicon to find out its possible domains, and thus gain some understanding of its meaning. The lexicon is implemented as an auxiliary relation of the following form: LEXICON=(TOKEN,RELATION,ATTRIBUTE). An example of a lexicon tuple is (SMITH,STUDENT,S-NAME), which states that the data token SMITH appears in attribute S-NAME of relation PERSON.

The THESAURUS relation stores synonym information about database tokens, associating various nondatabase tokens with database tokens. It is implemented as an auxiliary relation of the following form: THESAURUS =(WORD,TOKEN). The domain of words and the domain of tokens must be disjoint. Examples of thesaurus tuples are (STUDENTS,STUDENT), or (BOB,ROBERT), which state that the word STUDENTS should be understood as the metadata token STUDENT and the word BOB should be understood as the data token ROBERT.

D. An Overview of FLEX

Fig. 3 illustrates the overall architecture of FLEX. Initially, the user composes a query in a simple editor. When the user presses the *submit* button, the contents of the editor buffer are transferred to the parser. If the parser succeeds in parsing the input, it is passed to the query processor. If the parser fails, the input is piped through a sequence of three mechanisms (the corrector, the synthesizer, and the browser). Each of these mechanisms attempts to interpret the input. If a mechanism fails, it passes the input to the next mechanism; if the final mechanism fails, then FLEX gives up. If a mechanism succeeds in producing one or more interpretations (possibly after a brief clarification dialogue), it presents them to the user. If the user accepts an interpretation, it is copied back into the editor buffer, where it can be refined before resubmission. If the user does not accept any of the interpretations, the input is passed to the next mechanism. If a processed query returns a nonempty answer, it is displayed to the user. If the answer is empty, the input is passed to the query generalizer. The generalizer will suggest related queries that have nonempty answers (or it will point out erroneous presuppositions). If the user accepts one of these queries, it is copied back into the editor buffer; otherwise, processing of this input is terminated.

III. The Parser and the Corrector

When the user submits his input for processing, it is transferred to the parser. If parsing succeeds, the query is transferred to the processor. Therefore, the processing of perfect queries is not different than in any other query language interface.

As suggested earlier, the accuracy and specificity of the interpretations of FLEX corresponds to the correctness and well-formedness of its input. In this respect, the parser handles only input which is correct, and its interpretations are all accurate.

If parsing fails, the input is transferred to the query corrector. The corrector applies a set of transformations to try and salvage the query. Thus, the corrector handles input which is slightly imperfect, and its interpretations are mostly accurate.

A. Principles

Parsing begins with synonym substitution. The parser searches each input word in the THESAURUS relation. Any word that appears in the first attribute (WORD) is substituted by the corresponding word in the second attribute (TOKEN). Since the domains of words and tokens in the thesaurus are disjoint, only nondatabase tokens can be replaced. Therefore, substitutions may be considered safe, and remain in effect for the duration of processing.

The parser then checks that the input is correct both syntactically and semantically. The syntactic analysis verifies that the input is indeed a sentence in the query language (as defined by a grammar). The semantic analysis verifies that the sentence is meaningful (as defined by

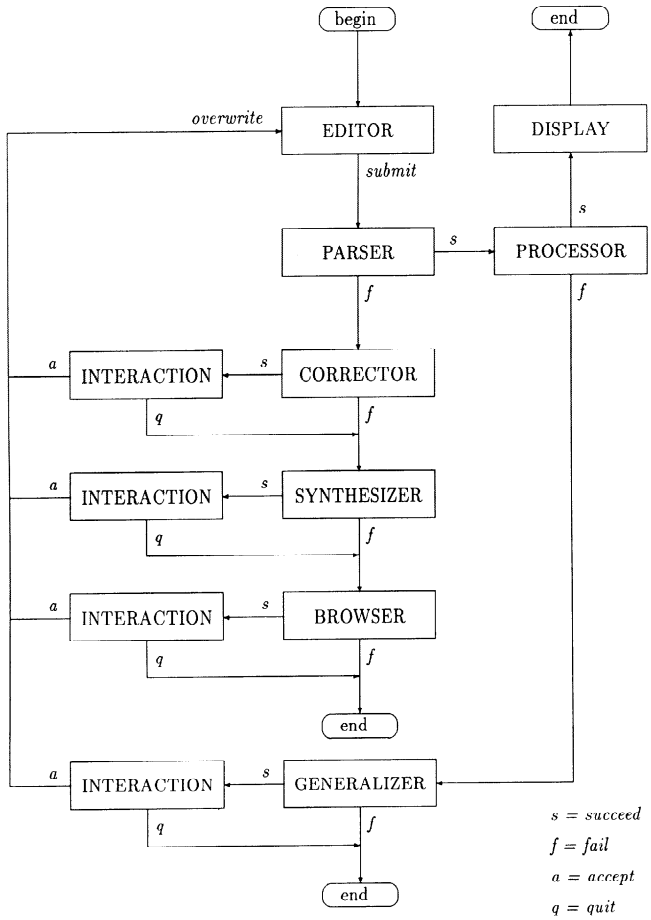


Fig. 3. Overall architecture of FLEX.

the database at hand). These two analyses are performed sequentially, and parsing is successful only if both succeed.

Correction is performed in two corresponding phases: if the syntactic analysis fails, the input is subjected to syntactic corrections; if the semantic analysis fails, the input is subjected to semantic corrections.

Correction is done without any interaction with the user. At the end of the entire correction process, the modified query is presented to the user for approval or further editing. The corrector requires that the input includes the keywords **retrieve** and **from** (in this order). To assure that only input which is slightly imperfect is corrected, the total number of transformations applied to the input is monitored. When a predefined number is reached, the corrector considers the input to be beyond salvation, and passes it to the next mechanism.

B. Corrective Transformations

1) Syntactic Correction: The transformations applied in this phase are intended to coerce the input into the syntax of the language. Clearly, among the three clauses of the retrieve statement, **retrieve, from,** and **where,** the latter is the most demanding, and most of the transformations are applied to this clause. They include:

1) Insert the default comparator = where a comparator is expected (i.e., between two attributes or between an attribute and a value), possibly replacing the current input token in this position.

2) Insert the default connector **and** where a connector is expected (i.e., between two subexpressions), possibly replacing the current input token in this position.

3) Supply missing parentheses, according to a default scheme.

4) Discard certain bad input tokens.

The other two clauses are essentially lists of attributes or relations. The transformations applied to these clauses include:

1) Accept list delimiters other than commas.

2) Discard bad list elements.

2) Semantic Correction: Verification of proper semantics is done against the database at hand. The semantic analysis includes five checks:

1) For each relation referenced in the query there should be a database relation by that name.

2) Each relation referenced in the **retrieve** or **where** clauses should be listed in the **from** clause.

3) For each qualified attribute referenced in the query (i.e., an attribute prefixed by a relation) there should be an attribute by that name in the qualifying relation.

4) For each unqualified attribute referenced in the query, there should be one database relation with that attribute.

5) Every two attributes that are compared in the **where** clause must be from the same domain; if an attribute is compared to a value, the value must be from the domain of the attribute.

First, the corrector discards any unrecognized relations. This may shorten the list of relations in the **from** clause, and it may remove the qualifiers of attributes in the **retrieve** and **where** clauses.

Next, the corrector attempts to qualify any unqualified attributes. If an unqualified attribute A appears in only one database relation R, then R is the qualifier of A. If A appears in several database relations, then the corrector attempts to infer a single one by elimination: relations that are not listed in the **from** clause are eliminated, a term $A\theta R.A$ in the **where** clause eliminates R because it would introduce a self-comparison, and a term $R.A = S.A$ eliminates either R or S, because both yield the same result. An unqualified attribute in the **retrieve** clause which does not appear in any database relation is discarded.

Next, the corrrector considers badly qualified attributes. A badly qualified attribute $R.A$ may be corrected either by substituting R by a relation that includes A, or by substituting A by another attribute of R. This set of possible substitutions is reduced by selecting relation substitutes only from the list of **from** relations, by selecting relation substitutes that would not introduce any self-comparisons in the **where** clause (e.g., a term $R.A\theta S.A$ eliminates the relation substitute S), and by selecting attribute substitutes that would not introduce any domain conflicts in the **where** clause (e.g., a term $R.A\theta S.B$ eliminates any attribute substitute whose domain is different from the domain of B).

Next, the corrector considers domain conflicts. A conflicting comparison $R.A\theta S.B$ may be corrected either by substituting A by an attribute of R that has the same domain as B, or by substituting B by an attribute of S that has the same domain as A.[1] Methods similar to those mentioned above are used to reduce the number of possible corrections.

Finally, the corrector considers relations that are referenced in the **retrieve** or **where** clauses, but are not listed in the **from** clause (note that such references may have been added by the corrector). These are simply added to the **from** clause.

If any of these semantic corrections fails (i.e., the corrector cannot infer a unique correction), the input is assumed to be beyond salvation.

C. Example

Consider the following input:

```
RETRIEVE S-NAME AND MAJOR;
FROM STUDENTS AND COURSES;
WHERE STUDENTS.S-NAME=S-NAME,
ENROLLMENT.C-NO=COURSES.NUMBER,
AND COURSES.D-NAME="CS";
```

The initial synonym substitution pass replaces STUDENTS with STUDENT, COURSES with COURSE, and CS with COMPSCI. Note that synonyms include both data and metadata. We have

```
RETRIEVE S-NAME AND MAJOR;
FROM STUDENT AND COURSE;
WHERE STUDENT.S-NAME=S-NAME,
ENROLLMENT.C-NO=COURSE.NUMBER,
AND COURSE.D-NAME="COMPSCI";
```

The syntactic analysis detects the keywords **retrieve, from,** and **where,** and applies these corrections. The keyword **and** and the trailing semicolon are removed from the **retrieve** and **from** clauses, and commas are inserted between the elements of these lists. The commas separating the terms of the **where** clause and its terminating semicolon are discarded, and the connector **and** is inserted between the first two terms. We have

```
RETRIEVE S-NAME, MAJOR
FROM STUDENT , COURSE
WHERE STUDENT.S-NAME=S-NAME
AND ENROLLMENT.C-NO=COURSE.NUMBER
AND COURSE.D-NAME="COMPSCI"
```

The semantic correction begins by qualifying the attributes S-NAME and MAJOR: S-NAME appears in both STUDENT and ENROLLMENT, but since the former would introduce a self-comparison in the **where** clause, it is qualified by the latter; MAJOR is qualified by STUDENT, which is the only relation to include it. Next, the bad qualification COURSE.NUMBER is replaced by COURSE.C-NO, C.NO being the only attribute of COURSE with the same domain as ENROLLMENT.C-NO. The corrected query is now formatted and displayed to the user:

```
retrieve ENROLLMENT.S-NAME,STUDENT.MAJOR
from STUDENT, COURSE, ENROLLMENT
where STUDENT.S-NAME=ENROLLMENT.S-NAME
    and ENROLLMENT.C-NO=COURSE.C-NO
    and COURSE.D-NAME="COMPSCI"
```

The user may now either *accept* this query or *abandon* the process. In the former case, the query is copied back into the editor, where the user may *refine* it before resubmitting it. In the later case, the corrector transfers the input to the next mechanism.

D. Related Research

Techniques for handling input errors have been implemented in compilers for general programming languages (for example, see [3, pp. 226–227] or [1, pp. 164–165]. A common technique involves augmenting the grammar with "error productions" that parse erroneous input. This technique is usually applied to allow the compiler to *recover* and continue its analysis (after generating appropriate error diagnostics). The FLEX corrector applies a similar technique in its syntactical correction phase.

In general, correcting semantic errors in large programs is considered to be both risky and expensive. The error correction capabilities of FLEX are a consequence of the relative simplicity of the language and the restricted semantics provided by the particular database being accessed.

IV. The Synthesizer

If the input does not exhibit sufficient syntactic and semantic structures to be salvaged by the query corrector, or if the user rejects the corrected query, then FLEX engages its query synthesizer.

The synthesizer treats the input as an unstructured set of words. The words that it recognizes are synthesized into proper queries. These queries are then presented to the user as educated guesses, and may be subject to further refinement by the user.

Thus, compared to the corrector, the synthesizer handles input which is less well-formed (a set of words), and its interpretations are less accurate (educated guesses.)

Indeed, it is possible to use FLEX as an interpreter of a very simple language, where the user provides sets of database tokens, and (after approving the suggested interpretations) is presented with the corresponding database output. Such input will fail all the mechanisms that precede the synthesizer.

A. Principles

The universe of recognized words is defined by the database at hand, as its set of database tokens (i.e., the union

[1] The corrector does not attempt to find another relation R' that includes an attribute A with the same domain as B, or another relation S' that includes an attribute B with the same domain as A, because the data model requires that attributes appearing in more than one relation have the same domain.

of its data and metadata elements). Words which are not recognized are discarded.

In general, tokens which are elements of the metadata would be interpreted as *requests*, while tokens which are elements of the data would be interpreted as *qualifiers*. For example, in the input "RETRIEVE STUDENT AND MAJOR FOR CS101," the words STUDENT and MAJOR are elements of the metadata, and would be understood as requests for data of those types; the word CS101 is an element of the data, and would be understood as a qualifier to help identify the data requested; the words RETRIEVE, AND, FOR are not database tokens, and would be discarded.

By its very nature, a set of tokens provides only fragmented information. The goal of the synthesizer is to connect these individual requests and qualifiers into a meaningful database query. To assist in this task the synthesizer represents the information stored in the dictionary as a graph called the *schema graph*.

The schema graph has a node for each relation, for each attribute, and for each domain (*relation nodes*, *attribute nodes*, and *domain nodes*, respectively). Each domain node is connected with edges to all the nodes of the attributes that draw their values from this domain. Each attribute node is connected with edges to all the relation nodes that include this attribute. Note that schema graphs are not necessarily connected and may have cycles.

The schema graph for the example is shown in Fig. 4. Note that this database definition is cyclic; for example, the association between a student and a course may be either that the student is enrolled in the course, or that the course is offered by the department in which he majors.

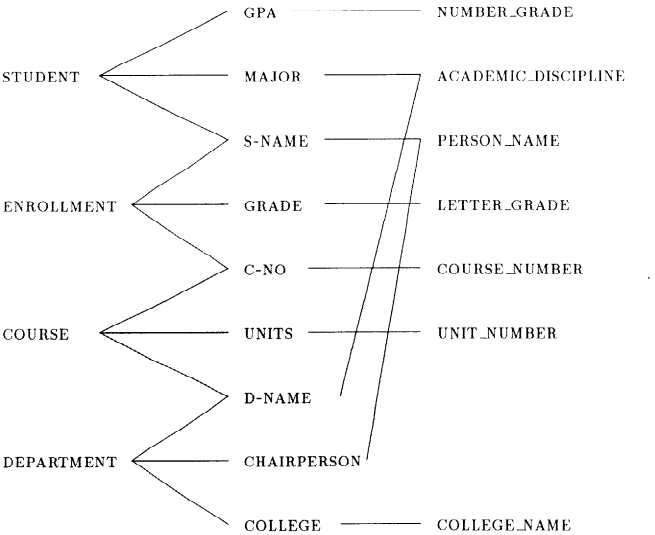


Fig. 4. Schema graph for database UNIVERSITY.

B. The Synthesis Procedure

The problem of synthesizing a formal query from given input is translated into a graph problem. Roughly, the input is modeled by a set of nodes in the schema graph, the nodes are then connected into a subgraph, and the subgraph is translated into a query. Thus, the problem of synthesizing a query can be divided into three subproblems: 1) how to determine the nodes that correspond to the given input 2) how to connect the nodes into a subgraph, and 3) how to translate this subgraph into a query.

To demonstrate this procedure, assume a user who is aware that the database contains information on students, courses, and enrollments, and would like to find out the names and majors of the students who are enrolled in the course CS 101. However, this user can only utter something like "RETRIEVE STUDENT AND MAJOR FOR CS101."

1) From Words to Nodes. In general, for each recognized word one node in the schema graph is selected. First, the word is searched in the dictionary. If the word is a relation name, an attribute name, or a domain name, then the corresponding relation node, attribute node, or domain node is selected. If the word is not found in the dictionary, it is searched in the lexicon. If it is found (i.e., it is a data token), then the node that corresponds to its domain is selected. Data tokens that belong to more than one domain are *ambiguous*. Such ambiguities are resolved by issuing to the user a request for clarification: the request displays the ambiguous word, along with its possible domains, and the user is asked to select the correct domain. If the word is not found in the lexicon, it is discarded.

Recall the example input "RETRIEVE STUDENT AND MAJOR FOR CS101". The words STUDENT and MAJOR select the corresponding relation and attribute nodes with these names, the word CS101 selects the domain node COURSE_NUMBER, and the words RETRIEVE, AND, and FOR are discarded.

2) From Nodes to Subgraph. To connect the selected nodes in the schema graph into a subgraph that spans them, the synthesizer applies an iterative procedure that finds an optimal (shortest) path between a given node and a set of nodes that have already been connected in a previous step.

It begins by ordering the selected nodes (possibly, by the order of the corresponding words in the input) and it *marks* the first node. It then considers the next node, and searches for the shortest path between this new node and the set of marked nodes (initially, this set includes only the first node). All the nodes of the path are marked, and the next selected node which is still unmarked is considered. This procedure continues until all the selected nodes have been marked. When it terminates, a connected subgraph is available. Obviously, this subgraph is always a tree.

Recall that the example input "RETRIEVE STUDENT AND MAJOR FOR CS101" selected three nodes: the relation node STUDENT, the attribute node MAJOR, and the domain node COURSE-NUMBER. First, MAJOR is connected to STUDENT with a single edge; then, COURSE-NUMBER is connected to this subgraph with a path that goes through C-NO, ENROLLMENT, and S-NAME. This subgraph is shown in Fig. 5.

When this procedure terminates, it is possible to have a subgraph with a domain node which is not connected to

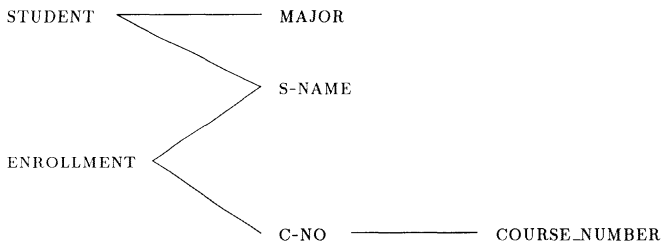


Fig. 5. Connected subgraph.

any attribute node (for example, when the only selected node is a domain node), or an attribute node which is not connected to any relation node (for example, when the only selected node is an attribute node.) In the former case, the domain node is connected to an associated attribute node, and in the latter case the attribute node is connected to an associated relation node. If there are several attribute nodes that are associated with the domain node, or if there are several relation nodes that are associated with the attribute node, then the user is asked to select one.

3) From Subgraph to Query: The subgraph is now transformed into a query as follows (the subsequent discussion is concerned only with the subgraph).

For each attribute node A:

- If A was selected by a metadata token, then $R.A$ is added to the **retrieve** clause, where R is one of A's adjacent relation nodes.
- If A has several adjacent relation nodes $R_1, \ldots, R_n$, then the selection phrase ($R_1.A = R_2.A$ **and** $\ldots$ **and** $R_{n-1}.A = R_n.A$) is conjoined to the **where** clause.

For each relation node R:

- R is added to the **from** clause.
- If R was selected by a metadata token, and the **retrieve** clause does not include any of its attributes, then all of R's attributes are added to the **retrieve** clause. If R was selected by a metadata token, and the **retrieve** clause includes some of its attributes, then only the key attributes of R are added to the **retrieve** clause.

For each domain node D:

- If D was selected by a metadata token, then $R.A$ is added to the **retrieve** clause, where A is one of D's adjacent attribute nodes and R is one of A's adjacent relation nodes.
- If D was selected by data tokens $C_1, \ldots, C_n$, then the selection phrase ($R.A = C_1$ **or** $\ldots$ **or** $R.A = C_n$) is conjoined to the **where** clause, where A is one of D's adjacent attribute nodes, and R is one of A's adjacent relation nodes.
- If D has several adjacent attribute nodes $A_1, \ldots, A_n$, which have adjacent relation nodes $R_1, \ldots, R_n$, then the selection phrase ($R_1.A_1 = R_2.A_2$ **and** $\ldots$ **and** $R_{n-1}.A_{n-1} = R_n.A_n$) is conjoined to the **where** clause.

Thus, an attribute mentioned in the input is interpreted as a request to retrieve that attribute. A relation mentioned in the input is interpreted as a request to retrieve its key attributes (if other attributes of this relation are mentioned in the input), or as a request to retrieve all its attributes (if none of its attributes are mentioned in the input). For example, "COURSE SMITH" will retrieve full details on the courses in which Smith is enrolled, while "COURSE UNITS SMITH" will retrieve only the course numbers and units of these courses. A domain mentioned in the input is interpreted as a request to retrieve an attribute of this domain.

The **where** clause is a conjunction of selection phrases. Each phrase either joins two relations or binds an attribute to a value (or to one of several values). For every attribute node that is adjacent to several relation nodes, or a domain node that is adjacent to several attribute nodes, the corresponding relations are joined. For every domain node that was marked by data tokens, the adjacent attribute is bound to the data tokens.

Finally, every relation whose node is in the subgraph is necessary for processing this query, and is therefore added to the **from** clause.

When applied to the subgraph of Fig. 5, this procedure synthesizes the following query:

retrieve S-NAME, MAJOR
from STUDENT, ENROLLMENT
where STUDENT.S-NAME=ENROLLMENT.S-NAME
  and ENROLLMENT.C-NO="CS101"

Its answer is

S-NAME	MAJOR
KLEIN	COMPSCI
SMITH	MATH

We note that there are several additional minor corrections that may be applied to queries generated by this procedure.

C. Alternative Interpretations

The synthesized query is suggested to the user. The user may then either *accept* it, *reject* it, or *abandon* the process. In the former case, this query is copied back into the editor, where the user may *refine* it before resubmitting it. In the latter case, the synthesizer terminates its attempts and transfers the input to the next mechanism. If the user rejects the query, the synthesizer tries to synthesize an alternative query. This is done by repeating steps 2 and 3 of the synthesis procedure. The synthesizer attempts to span the given nodes with a different subgraph, which serves as the basis for a new query.

As another example, consider the input "STUDENT MATH". Its tokens select the relation node STUDENT and the domain node ACADEMIC_DISCIPLINE. The shortest path that connects these nodes goes through the attribute node MAJOR. This subgraph yields the following query that lists all information on the students who are Math majors:

retrieve S-NAME, MAJOR, GPA
from STUDENT
where STUDENT.MAJOR="MATH"

Its answer is

S-NAME	MAJOR	GPA
BROWN	MATH	2.6
SMITH	MATH	3.2

If this query is rejected, the synthesizer will connect the selected node with an alternative path that goes through the attribute node MAJOR, the relation node STUDENT, the attribute node S-NAME, the relation node ENROLLMENT, the attribute node C-NO, the relation node COURSE, the attribute node D-NAME, and the domain node ACADEMIC_DISCIPLINE. It yields the following query that lists all information on the students who are enrolled in Math courses:

retrieve S-NAME, MAJOR, GPA
from STUDENT, ENROLLMENT, COURSE
where STUDENT.S-NAME=ENROLLMENT.S-NAME
and ENROLLMENT.C-NO=COURSE.C-NO
and COURSE.D-NAME="MATH"

Its answer is

S-NAME	MAJOR	GPA
BROWN	MATH	2.6
KLEIN	COMPSCI	2.8
SMITH	MATH	3.2

D. Related Research

This method of synthesizing queries from a set of tokens recalls work on the problem of inferring database joins automatically. That problem may be stated as follows: given a set of database attributes, derive a relation that combines these attributes. If a unique relation may always be derived, then a query language may be designed that relieves its users from navigating within relations, thus achieving higher independence from the logical structure of the database.

One approach to this problem, known as the *universal relation approach*, is to form the natural join of all the relations of the database, and then project on the given attributes [17]. There are several problems with this approach [13]. One problem is that all database attributes must have different names. An even more severe problem is that if the database definition is *cyclic* (i.e., two attributes may be connected through different sequences of joins), then this procedure may yield unnatural results. A possible solution is to define databases that do not include cycles; however, this may lead to complexities of design and replication of information, both contrary to the very purpose of databases.

A variation of this approach, which addresses these problems, is to incorporate *maximal objects* into the definition of the database [18]. Intuitively, each maximal object is a derived relation that represents a unique meaningful connection among its attributes. The given attributes are then projected from every maximal object that contains them, and the union of the answers is formed. This approach is the basis for the relational database System/U [14]. Except for the additional requirement to predefine the maximal objects, a major drawback of this approach is that the final answer may combine tuples which represent different connections of the given attributes.

Another approach for dealing with the presence of alternative connections, is to adopt a criterion of optimality. Usually, the given attributes select certain components in a graph that represent the definition of the database, and the preferred connection is the one that corresponds to a minimal subgraph that spans these components. Obviously, such subgraphs are always trees, and the problem is known as the *Steiner tree* problem [8]. (This problem is a generalization of the *minimum spanning tree* problem.) This approach was taken by [6], [26], [15], and [20]. While it often yields satisfactory results, it has three drawbacks: first, Steiner trees are not necessarily unique, and there may be several such trees, each leading to a different query; second, the query intended by the user may correspond to a subgraph which is not necessarily minimal; and, third, finding Steiner trees is a problem known to be NP-complete.

The FLEX approach is a variation of the minimal subgraph approach, with several significant differences. 1) FLEX handles a wider variety of input tokens. While others consider attribute names only, FLEX considers tokens which are either relation names, attribute names, domain names, or data. 2) FLEX does not force a single interpretation upon its users. Its interpretations are offered as suggestions; if rejected, FLEX spans the nodes differently, and synthesizes alternative queries. 3) Users are allowed to refine the queries suggested by FLEX. Even when a suggested query is not the one intended, the necessary modifications are often minor, and are relatively easy to perform because the syntactical and semantical structures are now mostly in place.

V. The Browser

If the input does not contain any metadata tokens, or if the user rejects all the synthesized queries (or otherwise abandons the synthesis process), then FLEX engages its browser to construct a browsing request for one of the recognized data tokens.

The browsing request retrieves from the database all the information that is available on the selected topic, and displays it to the user in a single frame.

Thus, the browser handles input which is much less well-formed (one topic), and its interpretations are much less specific (everything that is known on that topic).

Indeed, it is possible to use FLEX as a browsing tool, where the user provides topics, and is presented with frames of information on these topics. Such input will fail all the mechanisms that precede the browser.

A. Principles

Using the dictionary and the lexicon, the browser views the entire database as a single network of objects.

All the occurrences of a particular data token t under

database attributes that are associated with the same domain d are considered collectively to be one *object* called $t(d)$. For example, the object MATH (ACADEMIC_DISCIPLINE) is assembled from occurrences of the token MATH under STUDENT.MAJOR, COURSE.D-NAME, and DEPARTMENT.D-NAME. Note that by using domain information, objects are guaranteed coherent semantics. For example, if BROWN occurred in the database both under attributes whose domain is ACADEMIC_DISCIPLINE and under attributes whose domain is COLOR_NAME, then two separate objects would be assembled: BROWN (ACADEMIC_DISCIPLINE) and BROWN (COLOR_NAME).

Object *relationships* are based on the functional dependencies that are known to exist among the database attributes. In each relation, every attribute is functionally dependent on the key attributes. Consequently, each data token is related through functional dependencies to other tokens in the tuples in which it occurs. Since each object combines all the occurrences of a particular data token in the database, the relationships of this object to other objects are based on all the relationships in which these occurrences participate. Note that this object may be the source of a functional dependency in one relation, and the target of a functional dependency in another.

Consider again the object MATH.[2] It occurs once in DEPARTMENT.D-NAME, and several times in STUDENT.MAJOR and COURSE.D-NAME. On the basis of these tuples, this object is related to six other objects: SCIENCE and FOX (functionally dependent on MATH in relation DEPARTMENT), SMITH and BROWN (functionally determining MATH in relation STUDENT), and MATH270 and MATH370 (functionally determining MATH in relation COURSE).

By concatenating the relation names and the attribute names involved in each functional dependency, meaningful names for the relationships can be obtained. For example, MATH and FOX are related via **is** D-NAME **of** DEPARTMENT **having** CHAIRPERSON, and MATH and BROWN are related via **is** MAJOR **of** STUDENT **having** S-NAME.

The complete list of relationships of MATH is

MATH	**is** D-NAME **of** DEPARTMENT **having** COLLEGE	SCIENCE
MATH	**is** D-NAME **of** DEPARTMENT **having** CHAIRPERSON	FOX
MATH	**is** MAJOR **of** STUDENT **having** S-NAME	BROWN
MATH	**is** MAJOR **of** STUDENT **having** S-NAME	SMITH
MATH	**is** D-NAME **of** COURSE **having** C-NO	MATH270
MATH	**is** D-NAME **of** COURSE **having** C-NO	MATH370

Part of the object network derived from the database of Fig. 2 is shown in Fig. 6. Note that all edges represent two-way relationships: MATH is related to FOX via **is**

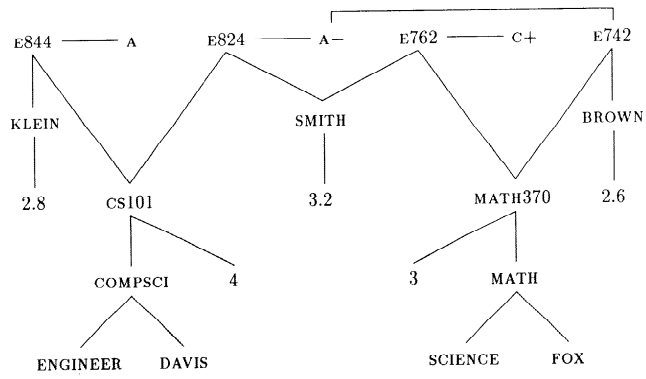


Fig. 6. Object network for database UNIVERSITY (part).

D-NAME **of** DEPARTMENT **having** CHAIRPERSON, and FOX is related to MATH via **is** CHAIRPERSON **of** DEPARTMENT **having** D-NAME.

Consider now the relation ENROLLMENT. Assume that its key attribute E-NO is removed, and, instead, the relation is keyed on the combination of S-NAME and C-NO. In this case, GRADE is functionally dependent on this combination. To define object relationships in such cases, it is necessary to introduce the notion of a *composite* object, which is a combination of objects. For example, the objects SMITH and MATH370 are combined to create the composite object (SMITH,MATH370). A composite object occurs in the database whenever its components appear *in the same tuple* of some relation under the *key attributes*. Composite objects need not have separate entries in the lexicon, since they can be located through the entries of their components.

Notice that the individual components of the key are themselves functionally dependent on the key. These so-called *trivial* dependencies are important, since they help establish relationships from components of the key to other data tokens in the tuple. For example, SMITH is related to both (SMITH,MATH370) and (SMITH,CS101), which, in turn, are related, respectively, to C+ and A−.

Let E-ID denote the combination (S-NAME, C-NO). The first two tuples of ENROLLMENT give rise to these six relationships:

SMITH	**is** S-NAME **of** ENROLLMENT **having** E-ID	(SMITH,MATH370)
SMITH	**is** S-NAME **of** ENROLLMENT **having** E-ID	(SMITH,CS101)
CS101	**is** C-NO **of** ENROLLMENT **having** E-ID	(SMITH,CS101)
MATH370	**is** C-NO **of** ENROLLMENT **having** E-ID	(SMITH,MATH370)
(SMITH,MATH370)	**is** E-ID **of** ENROLLMENT **having** GRADE	C+
(SMITH,CS101)	**is** E-ID **of** ENROLLMENT **having** GRADE	A−

[2]When a token belongs to only one domain, the domain will be deleted from the object name.

Obviously, except for their name, the new composite objects (SMITH,MATH370) and (SMITH,CS101) are identical to the previous simple objects E762 and E824.

B. Processing Requests

The browser extracts from the input a list of data tokens, and requests the user to select from this list a topic for a browsing request. The browser then locates the corresponding object in the object network and retrieves its immediate neighborhood: the adjacent relationships and objects. This neighborhood is structured as a *frame* of information on this topic, which is then presented to the user as all that is known about this topic.

The object network is not stored explicitly; only the portion required for the present request is constructed with several database accesses. First, the browser retrieves from the lexicon the attributes where the token occurs, and uses the dictionary to determine the various domains of the token. If the token belongs to more than one domain, then the browser requests the user to select the domain intended. For each attribute of the selected domain the browser issues a query to retrieve from the appropriate relation the tuples that have the given token in that attribute. In addition, the browser uses the dictionary to determine the keys of these relations. This information is used to determine the relationships in which the topic object participates and their names. The tuples returned from the various relations are then structured according to the relationships. The frame of the object MATH is shown in Fig. 7.

As mentioned earlier, by providing arbitrary new topics (i.e., data tokens), the user may use FLEX as a browsing tool. Note that this kind of access does not require any understanding of the underlying relational data model. If the user provides new topics from the data tokens mentioned in the current frame, he will be *navigating* in the object network. Note that this network view is never conveyed explicitly to the user, but will usually become apparent after repeated use.

C. Related Research

Browsers are tools for performing exploratory searches, often by naive or casual users. They are especially useful when either 1) the user is unfamiliar with the underlying data model, 2) the user is not proficient in the formal query language, 3) the user is not familiar with the contents and organization of the particular database being accessed, 4) the user has no preconceived retrieval target, or 5) the user cannot describe his retrieval target in terms that are understood by the system.

Browsers usually employ simple conceptual models and offer simple, intuitive commands. Often, the conceptual model is a network of some kind, and browsing is done by navigation: the user begins at an arbitrary point on the network (perhaps a standard initial position), examines the data in that "neighborhood", and then issues a new command to proceed in a new direction. While browsing, users gain insight into the contents and organization of the searched environment.

```
MATH is - - -
D-NAME of DEPARTMENT having COLLEGE SCIENCE
                            CHAIRPERSON FOX
D-NAME of COURSE having C-NO MATH270
                           MATH370
MAJOR of STUDENT having NAME BROWN
                           SMITH
```

Fig. 7. Frame of MATH.

Examples of browsers include Cattell's entity-based database interface [4]; SDMS, a graphical browsing tool for a "spatial" database system [9]; TIMBER, a browser for the INGRES relational database system [25], BAROQUE, a relational browser [19]; and KIVIEW, an object-oriented browser [24].

Because the tabular structures of relational databases do not lend themselves to a network representation, most browsers for relational databases are simply tools for scanning relations (either base relations or relations derived from queries), and therefore have only limited exploration capabilities. Browsing is confined to a single relation at a time, and it is not possible to browse across relation boundaries. If a user encounters a token while browsing, and wants to know more about it, he must determine first in what other relations this token might appear, then formulate a query, and resume browsing in the new relation.

Using a lexicon to represent a relational database as a network of objects to support browsing was first done in BAROQUE. Essentially, the FLEX browser is an adaptation of this approach to the larger framework of FLEX.

VI. The Generalizer

Consider a query to retrieve all non-Math majors enrolled in the course MATH 370 who received the grade A. As there is no enrollment for which the course is MATH 370, the student is not a Math major, and the grade is A, the database system returns an empty answer. This response, however, is misleading. Clearly, the author of this query seems to think that some non-Math majors are enrolled in MATH 370, and will conclude that none of them received the grade A. While, in fact, only Math majors are enrolled in this course.

A distinction is made between *genuine* empty answers, and these *fake* empty answers that actually reflect erroneous presuppositions on behalf of the user. Fake empty answers are misleading, as they are often mistaken for genuine empty answers (and may therefore be understood as reaffirmation of the user's presuppositions). Even genuine empty answers are unsatisfactory, because their information content amounts to a "shrug."

This is in contrast with human behavior, where the detection of erroneous presuppositions is common cooperative behavior (Chairperson: "Who are the non-Math majors in your class who received an A?" Professor: "All

the students in my class are Math majors''), and partial answers are usually provided when the query is legitimate, but does not have an answer (Chairperson: ''Who are the students in your class who received an A?'' Professor: ''Nobody; but Smith received an A−'').

Hence, empty answers are rarely satisfactory, and FLEX does not deliver them without further explanation and assistance. When a query that had been processed returns an empty answer, FLEX engages the generalizer. The generalizer attempts to infer the presuppositions of the user, test their correctness, and offer partial answers when appropriate.

A. Principles

The generalizer is based on these observations.

First, every query reflects a presupposition that the retrieval request it expresses is plausible (may possibly succeed). For example, a query to retrieve the non-Math majors who received an A in MATH 370 reflects a presupposition that there may be non-Math majors who received an A in Math 370. These are the kind of presuppositions handled by FLEX. Indeed, the correspondence between a query and the presupposition it reflects is so tight, that the terms will be used interchangeably.

Second, each presupposition is a source of more general (weaker) presuppositions. For example, from the presupposition that there may be non-Math majors who received an A in MATH 370, the presupposition that there may be non-Math majors who received at least a B in MATH 370 and the presupposition that there may be students who received an A in MATH 370 may be inferred. Presuppositions that are *minimally* more general than a given presupposition (i.e., are weaker by the smallest ''notch'' expressible in the system) will be called *immediate* generalizations.

Third, given two presuppositions (inferred from the same query), the user is more confident about the more general presupposition. For example, the user is more confident about the existence of non-Math majors who received at least a B in MATH 370, or the existence of students who received an A in MATH 370, than about the existence of non-Math majors who received an A in MATH 370.

Thus, while users expect that their queries may possibly have empty answers, they tend to be confident that every more general query would not have failed. Consequently, the following test is devised: When a query fails, its immediate generalizations are generated and attempted. If all succeed, it is an indication that the original empty answer was *genuine*; the answers to the generalizations may then be considered *partial* answers. If at least one of the immediate generalizations fails, it is an indication that the original empty answer was *fake*; each failed generalization reflects an erroneous presupposition.

Clearly, if one query is a generalization of another and both fail, then the erroneous presupposition behind the more specific query is insignificant. Hence, a failure is *significant*, only if all its generalizations succeed. The previous test is therefore continued until all significant failures are detected.

The test can now be described as follows: When a query fails, the set of significant failures is determined. If the only significant failure is the query itself, then the empty answer is genuine (and each of its generalizations is a partial answer); otherwise, the empty answer is fake (and each significant failure reflects an erroneous presupposition).

B. The Generalization Procedure

FLEX traps each query that returns a empty answer and passes it to the generalizer. To generalize this query, its **where** clause is converted to conjunctive normal form; i.e., a conjunction of terms, where each term is a disjunction of primitive terms, where a primitive term is a comparison between two attributes or between an attribute and a value (negations are removed by using complementary comparators). This conjunctive query is generalized into a set of queries by modifying a single primitive term at a time.

Primitive terms are either numeric or nonnumeric, depending on the type of their operands, as specified in the dictionary. To generalize a numeric term, such as GPA > 3.6, the *delta* specified in the dictionary is used to relax the comparison by one ''notch,'' in this example, GPA > 3.4. A nonnumeric term, such as MAJOR=''COMPSCI'', is generalized by replacing it with *true*. If a numeric comparison is relaxed beyond the *minimum* or *maximum* values of this domain, as specified in the dictionary, then it, too, is replaced with *true*. If a primitive term is replaced with *true*, then the value of the entire conjunct becomes true, and it may be removed.

As an example, consider this query to retrieve the students with GPA over 3.6, whose major is either Computer Science or Electrical Engineering.

```
Q0: retrieve S-NAME
    from STUDENT
    where STUDENT.GPA > 3.6
      and (STUDENT.MAJOR=''COMPSCI''
        or STUDENT.MAJOR=''ELECENG'')
```

Its **where** clause is already in conjunctive normal form, and the following two generalizations are derived:

```
Q1: retrieve S-NAME
    from STUDENT
    where GPA > 3.6
```

```
Q2: retrieve S-NAME
    from STUDENT
    where STUDENT.GPA > 3.4
      and (STUDENT.MAJOR=''COMPSCI''
        or STUDENT.MAJOR=''ELECENG'')
```

Q_1 omits the requirement on the major, and Q_2 relaxes the requirement on the GPA. Q_1 is generalized further by decreasing the GPA requirement to 3.4 (Q_3), and Q_2 is generalized further by omitting the requirement on the major

(Q_3), or by relaxing the GPA requirement to 3.2 (Q_4). This continues until the threshold GPA value reaches the minimum specified in the dictionary, at which point the GPA requirement is deleted altogether. Fig. 8 illustrates the complete hierarchy of queries, assuming a minimum GPA value of 3.0. The top three queries are as follows:

Q_7: **retrieve** S-NAME
from STUDENT
where GPA > 3.0

Q_8: **retrieve** S-NAME
from STUDENT
where STUDENT.MAJOR = ''COMPSCI''
or STUDENT.MAJOR = ''ELECENG'')

Q_9: **retrieve** S-NAME
from STUDENT

This generalization strategy is slightly different when the query involves several relations that are *joined* by the **where** clause. Assume a query that involves relations R_1, . . . , R_n, and consider a particular join between R_i and R_j. If a term that joins these two relations is removed, and it is the only connection (either directly or indirectly) between these two relations, then the query becomes ''disconnected,'' with its n relations separated into two disjoint subsets (one including R_i, the other including R_j). This query will now derive a relation from each of the subsets, and compute their product. Usually, such queries have little intuitive meaning. To avoid such queries, the generalization procedure is modified as follows.

A join term whose immediate generalization is *true*, and whose removal disconnects the query, is called a *disconnecting term*.[3] A disconnecting term is removed only when one of the disjoint subsets of relations it creates has only one relation, and no other terms involve this relation. In the new query, this relation is removed from the **from** clause, and all its attributes are removed from the **retrieve** clause.[4]

As an example, consider this query to retrieve the GPA and grade of Computer Science majors enrolled in MATH 370.

Q_0: **retrieve** S-NAME, GPA, GRADE
from STUDENT, ENROLLMENT
where STUDENT.MAJOR = ''COMPSCI''
and ENROLLMENT.C-NO = ''MATH370''
and STUDENT.S-NAME = ENROLLMENT.S-NAME

Its **where** clause is already in conjunctive normal form, and the following two generalizations are derived:

Q_1: **retrieve** S-NAME, GPA, GRADE
from STUDENT, ENROLLMENT
where ENROLLMENT.C-NO = ''MATH370''
and STUDENT.S-NAME = ENROLLMENT.S-NAME

[3]Note that if a join term is numeric, its generalization could not involve disconnection, unless the threshold values have already been reached.

[4]In the ''pathological'' case where the **retrieve** clause becomes empty, the *key* attributes of the relations addressed by the query (the relations in the other subset) are inserted into it.

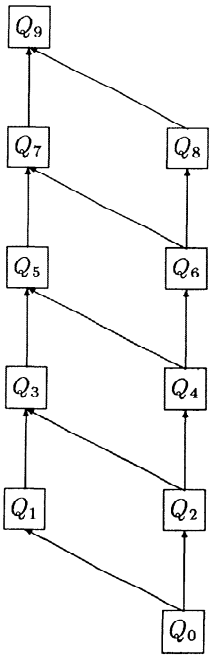


Fig. 8. Hierarchy of generalizations for first example.

Q_2: **retrieve** S-NAME, GPA, GRADE
from STUDENT, ENROLLMENT
where STUDENT.MAJOR = ''COMPSCI''
and STUDENT.S-NAME = ENROLLMENT.S-NAME

Q_1 omits the requirement on the major, and Q_2 omits the requirement on the course. By omitting the requirement on the course in Q_1, or the requirement on the major in Q_2, both queries are generalized to Q_3:

Q_3: **retrieve** S-NAME, GPA, GRADE
from STUDENT, ENROLLMENT
where STUDENT.S-NAME = ENROLLMENT.S-NAME

Q_1 and Q_2 can also be generalized by removing the join term. In Q_1 the relation STUDENT and the attribute GPA are removed from its **from** and **retrieve** clauses, yielding Q_4. Similarly, in Q_2 the relation ENROLLMENT and the attribute GRADE are removed from its **from** and **retrieve** clauses yielding Q_5.

Q_4: **retrieve** S-NAME, GRADE
from ENROLLMENT
where ENROLLMENT.C-NO = ''MATH370''

Q_5: **retrieve** S-NAME, GPA
from STUDENT
where STUDENT.MAJOR = ''COMPSCI''

Q_3 is generalized further by omitting the join term, and removing either the relation STUDENT and the attribute GPA, or the relation ENROLLMENT and the attribute GRADE, yielding Q_6 and Q_7:

Q_6: **retrieve** S-NAME, GRADE
from ENROLLMENT

Q_7: **retrieve** S-NAME, GPA
from STUDENT

Finally, Q_4 and Q_5 are generalized by omitting their **where** clauses altogether, also yielding Q_6 and Q_7. Fig. 9 illustrates these generalization relationships. Note that the generalizations Q_1 to Q_4, Q_2 to Q_5, Q_3 to Q_6, and Q_3 to Q_7 change the list of retrieved attributes.

The generalization procedure outlined above may be specified with a formal algorithm as follows. Assume that *gen*1 (q, i) is a subroutine that receives a query q and an index i and returns a new query, where the ith term of q has been substituted by an immediately more general term (recall that if the new term is *true*, then the entire conjunct is removed). Thus, *gen*1 takes care of generalizations on nondisconnecting terms. Assume that *gen*2 (q, r) is a subroutine that receives a query q and a relation name r and returns a new query, where all references to r have been removed (recall that it may be necessary to insert new **retrieve** attributes). Thus, *gen*2 takes care of generalizations on disconnecting terms. The following algorithm takes a given query Q_0 and generates a set of queries Q_1, . . . Q_m that are immediate generalizations of Q_0. Q_0 is assumed to have a **where** clause in conjunctive normal form with terms $t_1, \ldots, t_n$. If t_i is a join term, then $r_{i,1}$ and $r_{i,2}$ denote the two participating relations. Note that $Q_1, \ldots, Q_m$ may contain replications.

```
procedure gen(q);
m := 0;
for i := 1 to n do
begin
  if t_i is not a disconnecting term then
  begin
    m := m + 1;
    Q_m := gen1(Q_0, i);
  end
else
begin
  if t_i is the only term that references r_{i,1} then
  begin
    m := m + 1;
    Q_m := gen2(q, r_{i,1});
  end
  if t_i is the only term that references r_{i,2} then
  begin
    m := m + 1;
    Q_m := gen2(q, r_{i,2});
  end
end
end
```

C. Examples

In the first example, assume that Q_0 is presented to the database. As there are no students with GPA's over 3.6 in either Computer Science or Electrical Engineering, its answer is empty, and the query is passed to the generalizer.

In the first iteration, the generalization procedure derives Q_1 and Q_2 and submits them to the processor. Q_1 still fails, but Q_2 succeeds. Therefore, Q_1 is generalized further. In the second iteration, the generalization procedure derives Q_3 and submits it to the processor. It succeeds and the procedure terminates. Q_1 is a significant failure.

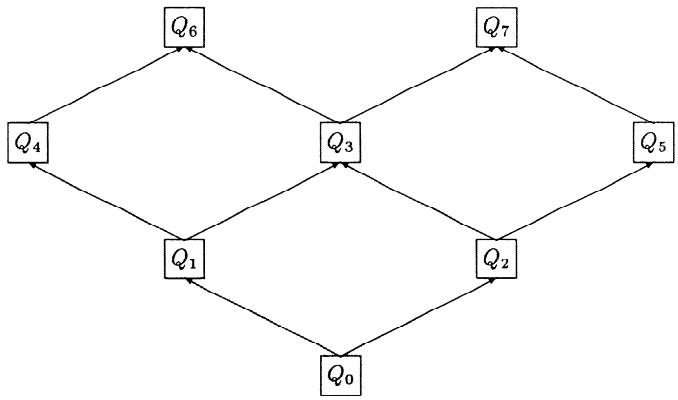


Fig. 9. Hierarchy of generalizations for second example.

In response to his original query, the user is presented with the message "Possible erroneous presuppositions—cannot answer even these more general queries:", followed by query Q_1, which retrieves the students with GPA over 3.6. In effect, the system is telling the user: "not only are there no students with GPA over 3.6 in these two departments, there are no such students in the entire college!"

In the second example, assume that Q_0 is presented to the database. As there are no Computer Science majors enrolled in MATH 370, its answer is empty, and the query is passed to the generalizer.

In the first iteration, the generalization procedure derives Q_1 and Q_2 and submits them to the processor. Both queries succeed and the procedure terminates. Q_0 is a significant failure.

In response to his original query, the user is presented with the message "No data matched—partial answers available:", followed by queries Q_1 and Q_2, which retrieve the information requested for all MATH 370 students, or for all enrolled Computer Science majors. The user can now submit any of these queries.

Finally, assume that in the second example the user misspells the major, typing "CS" instead of "COMPSCI".[5] Q_2 and Q_5 fail. Q_5 is a significant failure, and is presented as an erroneous presupposition. Since it retrieves information on students whose major is CS, the system is saying "there is no such major!".

D. Related Research

First to address the problem of empty answers was Kaplan [12], who designed and implemented a natural language interface to CODASYL databases, called CO-OP, which featured some of the conventions of cooperation in human discourse, including *corrective* responses that detect erroneous presuppositions and *suggestive* responses that anticipate followup queries. CO-OP trans-

[5]And assume that CS is not a synonym for COMPSCI.

forms each natural language query to an intermediate language, called Meta-Query Language (MQL). An MQL representation is a graph, whose connected subgraphs correspond to the presuppositions the user has made about the domain of discourse. When the initial query returns an empty answer, CO-OP tests each of these presuppositions, by translating the corresponding subgraph into a query in the formal language, and checking it in the database.

Providing quality responses to natural language queries that generate empty answers is also the topic of [16] (which also surveys other related works in cooperative interfaces). Closer to the framework of FLEX, several works have dealt with the problem of empty answers that are issued by typical database systems in response to formal language queries.

Janas [11] considers a family of predicate calculus queries, and shows how to generate a set of *predecessors queries* (queries whose predicates are satisfied whenever the predicate of the original query is satisfied) for a given query from this family. These queries are then checked against the database. The main technique for generating predecessor queries is to remove primitive terms from the predicate calculus expression. Referential integrity constraints are used to reduce the number of predecessor queries that must be checked.

Corella *et al.* [7] adapted these techniques to simple Boolean queries with a single existential variable, and implemented a cooperative front-end to a large bibliographical database.

These techniques are futher refined in [22] and [23], which define the concepts *generalization query*, *significant failure*, *genuine empty answer*, and *fake empty answer*, and then formulate a strategy for detecting significant failures, thereby determining whether an empty is genuine or fake, and generating, accordingly, either partial answers or erroneous presuppositions. The FLEX generalizer is based on these works. [22] also suggests using other data model features, such as subclass hierarchies, for generalizing queries. [23] extends the entire technique by testing the queries not only against data, but also against database knowledge (e.g., completeness assertions and integrity constraints).

VII. Implementation

FLEX was fully implemented as an interface to the database system INGRES [10]. It was written in the programming language C, on a Sun workstation running Unix. FLEX is purely a front-end interface; that is, it communicates with the database system only by issuing queries and receiving answers.

The current version of FLEX assumes a terminal with a standard display of 24 lines and 80 characters, and a standard keyboard with additional 16 programmable keys. FLEX defines three screens called *compose*, *query*, and *answer*, and provides three special keys for instantaneous switching between these screens. All three screens are structured similarly with a *text* window (for queries or answers) and a *menu* window (listing the commands for that particular screen). Initially, the user is in the compose screen. Its text window is an editor buffer; the user types his query into this window and edits it with simple commands shown in the menu window. When ready, the user presses a special key to copy the query to the text window of the query screen, and switches to that screen. The user may then submit the query by pressing another special key. Throughout the processing of his input, the user remains in the query screen. All requests for clarifications and all suggested interpretations are shown in windows that are overlayed on this screen. When the user accepts an interpretation, it replaces the contents of the text window. The user may then submit it immediately for processing, or copy it back to the text window of the compose screen, for further editing. Answers are shown in the text window of the answer screen, and can be scanned with the commands shown in the menu window of that screen.

The "knowledge base" used by FLEX consists of three auxiliary relations, that are stored along with the database itself: DICTIONARY, LEXICON and THESAURUS. The dictionary is used by every FLEX mechanism, the lexicon is used by the synthesizer and the browser, and the thesaurus is used by the corrector. The DICTIONARY relation is relatively small, the information it contains is fairly standard,[6] and it needs to be updated only when the definition of the database is changed. The LEXICON relation is more demanding in terms of size and maintenance. This relation should 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). 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. If the required storage is prohibitive, it is possible to implement the lexicon only in part, by inverting on selected domains only; tokens of other domains will not be recognized. The THESAURUS relation is different, in that its information cannot be extracted automatically from the database. It may be constructed gradually by the database owner, using a log of unrecognized words maintained by the system. While the thesaurus enhances the operation of FLEX, it is not as essential as the other two relations.

VIII. Conclusion

The most prominent design feature of FLEX is the smooth concatenation of several independent mechanisms, each capable of handling input of decreasing level of correctness and well-formedness. Each input is "cascaded" through this series of mechanisms, until an interpretation is found.

Consequently, FLEX may be viewed as an interface that *adapts* to the level of correctness and well-formedness of

[6] The *delta* parameter is, perhaps, the only exception.

its input (providing interpretations of corresponding accuracy and specificity). Due to this feature, FLEX can service satisfactorily users with different levels of expertise, and thus appeal to a more *universal* community of users.

This ability to adapt (which may also be regarded as *tolerance* towards incorrect input) is complemented with features of *cooperative* behavior, whereby empty answers are never delivered without explanation or assistance.

Tolerance and cooperation are achieved with only minimal interaction, avoiding excessively long dialogues, which tend to be tedious and discouraging. FLEX approaches its users mainly to determine the domain of an ambiguous token, or to select from a list of possible browsing topics. Both tasks are relatively short and simple.

By providing interpretations of ill-formed queries, FLEX also instructs its users in the proper application of the formal language. By providing alternative interpretations, and allowing them to be refined, FLEX reduces the risk of misinterpretations.[7]

FLEX can also be perceived as an interface that supports multiple languages, each with its own level of expressivity: a formal language, a language whose queries are sets of database tokens, and a language whose queries are individual topics. The mechanisms of FLEX would then be viewed, not as procedures for coping with incorrect formal queries, but as interpreters of these languages. Users may then deliberately submit queries in an "inferior" language; their input will flow through the interpreters of the "superior" languages, until it arrives at the intended interpreter, and generates the expected database request.

Work on FLEX is continuing. Current goals include: 1) extension of the retrieval language to include a fuller set of operators (e.g., aggregate operators, grouping operators); 2) improved presentation (e.g., employ a larger display that accommodates all three screens simultaneously, accept inputs via a "mouse"); and 3) various improvements to individual mechanisms. Some of the improvements being considered are: a) in the corrector: modify it to present the user with various alternative suggestions (currently, the corrector is the only mechanism that produces a single suggestion); b) in the browser and synthesizer: modify the lexicon to store *templates* of tokens (simple regular expressions) [2]; c) in the browser: consider methods that will limit or summarize the output; and d) in the generalizer: consider methods that will limit the number of the queries tested, for improved performance.

To obtain better feedback on the effectiveness of FLEX, we plan to experiment with it in a classroom environment. To maximize the benefits of this experiment, FLEX will be instrumented to record user inputs, system interpretations, and user reactions. In addition to the instructional potential of FLEX mentioned above, the inherent diversity of a student population, and the improvement of skills as a class progresses, suggest that an interface like FLEX could be a "perfect partner." It would provide more assistance to weak students, and less assistance to good students; more assistance in early stages (when inputs tend to be more erroneous), and gradually less and less assistance in later stages (when inputs tend to be more correct).

[7]Of course, the risk of misinterpretation is still present here, as in any process of interpretation (even a formal query may not convey the intentions of its author correctly).

Acknowledgment

The author wishes to thank Q. Yuan and S. Mathur for their assistance in the implementation and for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] B. W. Ballard, J. C. Lusth, and N. L. Tinkham, "LDC-1: A transportable, knowledge-based natural language processor for office environments," *ACM Trans. Office Inform. Syst.*, vol. 2, no. 1, pp. 1-25, Jan. 1984.

[3] P. Calingaert, *Assemblers, Compilers and Program Translation*. Potomac, MD: Computer Science Press, 1979.

[4] R. G. G. Cattell, "An entity-based database interface," in *Proc. ACM-SIGMOD Int. Conf. Management Data*, ACM, Santa Monica, CA, May 14-16, 1980, pp. 144-150.

[5] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "Sequel 2: A unified approach to data definition, manipulation, and control," *IBM J. Res. Develop.*, vol. 20, no. 6, pp. 560-575, Nov. 1976.

[6] C. L. Chang, "Finding missing joins for incomplete queries in relational data bases," Tech. Rep. RJ2145, IBM Research Lab., San Jose, CA, Feb. 1978.

[7] F. Corella, S. J. Kaplan, G. Wiederhold, and L. Yesil, "Cooperative responses to Boolean queries," in *Proc. IEEE Comput. Society First Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 24-27, 1984. Washington, DC: IEEE Computer Society, pp. 77-85.

[8] S. Even, *Graph Algorithms*. Potomac, MD: Computer Science Press, 1979.

[9] C. Herot, "Spatial management of data," *ACM Trans. Database Syst.*, vol. 5, no. 4, pp. 493-513, Dec. 1980.

[10] *SunINGRES Manual Set*, Sun Microsystems, Mountain View, CA, Release 5.0 Part Number 800-1644-01, 1987.

[11] J. M. Janas, "Towards more informative user interfaces," in *Proc. Fifth Int. Conf. Very Large Data Bases*, ACM, Rio de Janerio, Brazil, Oct. 3-5, 1979, pp. 17-23.

[12] S. J. Kaplan, "Cooperative responses from a portable natural language query system," *Artif. Intell.*, vol. 19, no. 2, pp. 165-187, Oct. 1982.

[13] W. Kent, "The universal relation revisited," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 644-648, Dec. 1984.

[14] H. F. Korth, G. M. Kuper, J. Feigenbaum, A. van Gelder, and J. D. Ullman, "System/U: A database system based on the universal relation assumption," *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 331-347, Sept. 1984.

[15] W. Litwin, "Implicit joins in the multidatabase system MRDSM," in *Proc. IEEE Comput. Soc. 9th Int. Comput. Software Appl. Conf.*, Chicago, IL, Oct. 9-11, 1985, pp. 495-504.

[16] W-S. Luk, M. Kao, and N. Cercone, "Providing quality responses with natural language interfaces: The null value problem," *IEEE Trans. Software Eng.*, vol. SE-14, no. 7, pp. 959-984, July 1988.

[17] D. Maier, *The Theory of Relational Databases*. Rockville, MD: Computer Science Press, 1983.

[18] D. Maier and J. D. Ullman, "Maximal objects and the semantics of universal relation databases," *ACM Trans. Database Syst.*, vol. 8, no. 1, pp. 1-14, Mar. 1983.

[19] A. Motro, "BAROQUE: A browser for relational databases," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 2, pp. 164-181, Apr. 1986.

[20] ——, "Constructing queries from tokens." in *Proc. ACM-SIGMOD Int. Conf. Management Data*, Washington, DC, May 28-30, 1986. New York: ACM, pp. 120-131.

[21] ——, "The design of FLEX: A tolerant and cooperative user interface

to databases," in *Proc. Second Int. Conf. Human-Comput. Interaction*, Vol. 2, Honolulu, HW, Aug. 10-14, 1987, p. 583-590, The International Commission on Human Aspects in Computing, Geneva, Switzerland. Conference proceedings available as Volumes 10A and 10B in the series Advances in Human Factors/Ergonomics, Elsevier Science Publishers.

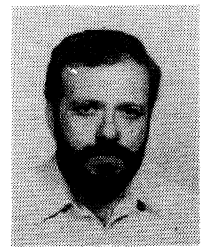
[22] ——, "Query generalization: A technique for handling query failure," in *Proc. First Int. Workshop Expert Database Syst.*, Kiawah Island, SC, Oct. 24-27, 1984, pp. 314-325, Institute of Information Management, Technology and Policy, Univ. of South Carolina, Columbia, SC.

[23] ——, "SEAVE: A mechanism for verifying user presuppositions in query systems," *ACM Trans. Office Inform. Syst.*, vol. 4, no. 4, pp. 312-330, Oct. 1986.

[24] A. Motro, A. D'Atri, and L. Tarantino, "The design of KIVIEW: An object-oriented browser," in *Proc. Second Int. Conf. Expert Database Syst.* Tysons Corner, VA, Apr. 25-27, 1988, pp. 17-31, George Mason University, Fairfax, VA.

[25] M. Stonebraker and J. Kalash, "TIMBER: A sophisticated database browser," in *Proc. Eighth Int. Conf. Very Large Data Bases*, Mexico City, Mexico, Sept. 8-10, 1982. Los Altos, CA: Morgan-Kaufmann, pp. 1-10.

[26] J. A. Wald and P. G. Sorenson, "Resolving the query inference problem," *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 348-368, Sept. 1984.



Amihai Motro received the B.Sc. degree in mathematical sciences from Tel Aviv University, Tel Aviv, Israel, in 1972, the M.Sc. degree in computer science from the Hebrew University, Jerusalem, Israel, in 1976, and the Ph.D. degree in computer and information science from the University of Pennsylvania, Philadelphia, in 1981.

Since 1981 he has been with the faculty of the Computer Science Department at the University of Southern California, Los Angeles. His main research area is data and knowledge management, in particular, intelligent user interfaces to databases, query languages for data and knowledge, and integration of databases. He is also interested in operating systems, and has worked for several years as a systems programmer.

Dr. Motro is a member of the Association for Computing Machinery and the IEEE Computer Society.