

The Design of FLEX: A Tolerant and Cooperative User Interface to Databases

Amihai Motro

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

Abstract

FLEX is a user interface to relational databases that is *tolerant* of incorrect input. FLEX never rejects a query; instead, it adjusts to the level of technical expertise its users seem to possess (as judged from their input). In particular, FLEX understands formal queries; salvages incorrect queries if they include enough clues on their intended meaning; suggests educated guesses if it recognizes metadata tokens in the input; or else, it issues browsing requests for recognized data tokens. FLEX is also *cooperative*. It never delivers null results without explanation and assistance. By following up each failed query with a set of more general queries, FLEX determines whether a null result is *genuine* (it then suggests related queries that have non-null results), or whether it reflects *erroneous presuppositions* on behalf of the user (it then explains them).

1 Introduction

The most common method for retrieving information from databases is through *formal query interfaces*; i.e., interfaces that require their users to define exact retrieval requests in a formal language. While such interfaces can prove to be most efficient, they also require considerable technical skills and preparatory knowledge. In particular, they require

- Familiarity with the principles by which data is organized (the *data model*).
- Proficiency in the procedures for specifying retrieval requests (the *data language*).
- Familiarity with the structure (*schema*) of the particular database being accessed.
- Familiarity with the contents (*semantics*) of the database.
- Clear retrieval targets (e.g., it is impossible to retrieve something "interesting" or "suitable").
- Ability to define the targets as required by the system (e.g., to retrieve the meaning of a word from a dictionary database, it is necessary to know its spelling).

In the absence of even some of the necessary expertise (skill or knowledge), formal retrieval can become very inefficient and frustrating.

We shall refer to users that do not possess the necessary expertise as *naive* users. To help naive users access databases several approaches have been attempted, including: form-based interfaces, in which queries are specified by entering information in predefined

screens (e.g., [INGRES 1984], [ORACLE 1983], [UNIFY 1983]; browsing interfaces, that enable users to explore databases with a small set of intuitive commands (e.g., [Herot 1980], [Stonebraker and Kalash 1982], [DBASE 1984], [Motro 1986a]); graphic-based interfaces, that use menus, icons and pointing devices to replace some of the literal communication (e.g., [McDonald and Stonebraker 1975], [Wong and Kuo 1982], [Fogg 1984]), and natural language interfaces (e.g., [Codd et al 1978], [Hendrix et al 1978], [Harris 1984]).

In general, these tools indeed make it easier for naive users. However, form interfaces, browsers, and graphic interfaces generally sacrifice the retrieval power of formal query languages, and, quite often, using them may become tedious. The advantage of natural language interfaces is that they can service users with different levels of expertise, handling formal requests, as well as vague or even ungrammatical requests. Unfortunately, current natural language interfaces have two major drawbacks: they require enormous investment to capture the knowledge that is necessary to understand user requests, and even the best systems are prone to errors.

In this article we report on research to develop a user interface to databases, which may be used satisfactorily by experts as well as novices. This interface, called FLEX, is based on a formal query language, but is *tolerant* of incorrect input. It never rejects queries; instead, it adjusts to the level of technical expertise its users seem to possess (as judged from their input). FLEX is also *cooperative*. It never delivers null results without explanation or assistance. Hence, in some respects, FLEX exhibits behavior reminiscent of natural language interfaces.

A useful metaphor for the approach taken in FLEX is a salesperson who is approached by a customer. Customers going into a store have different levels of "preparedness", and a good salesperson should be able to classify each buyer promptly, and offer the appropriate treatment. For example,

- If the customer asks for something specific, the salesperson directs him to that item.
- If the customer asks for something specific that is not available, the salesperson shows him similar items.
- If the customer's request reflects erroneous presuppositions, the salesperson tries to set him straight.
- If the customer's request is incorrect, the salesperson nevertheless makes an effort to understand it (possibly by means of a dialogue).
- If the customer's request does not make sense at all, the salesperson tries to infer a sensible request from recognized words.
- If the customer does not know (or cannot articulate) what he wants, the salesperson shows him around.

Similarly, FLEX incorporates several query processing mechanisms for processing requests of various levels of well-formedness. In correspondence with the salesperson strategy, FLEX includes these mechanisms:

- A formal query processor (PASS).
- A mechanism that detects erroneous presuppositions and suggests "alternatives" when queries cannot be satisfied (NULL).
- A mechanism that fixes incorrect queries safely (FIX).
- A mechanism that constructs default queries from recognized tokens (GUESS).
- A mechanism that issues browsing requests for recognized tokens (BROWSE).

2 The Architecture of FLEX

FLEX divides the screen of the data terminal into three major windows, called *compose*, *query* and *response*. *compose* is a user window, an elementary editor where the user enters queries. *query* and *response* are system windows: *query* displays the query being processed, and *response* displays its result. Upon initialization, the user is in the *compose* window. When he issues the command *process*, the contents of the *compose* window are copied to the *query* window, and processing begins.

The formal query processor PASS is engaged to check the syntax of the input. If the input is found to be incorrect, then a system component called FIX is engaged to salvage the query. If it determines that the query is fixable, then the system displays the message "Incorrect query ... fixing". If the query is not fixable, then a system component called GUESS is engaged to construct a query that approximates the intentions of the user. If it determines that a guess can be made, then the system displays the message "Incorrect query ... guessing". In either case a simple dialogue may follow, and eventually the new query is displayed in the *query* window. The user may either request to process it, or he may edit it further. If GUESS cannot even make a guess, then a system component called BROWSE is engaged to construct a browsing request for recognized input tokens. The system displays the message "Incorrect query ... browsing", a simple dialogue may follow, and eventually a browsing request is displayed in the *query* window. If approved, a frame of information is retrieved and displayed in the *response* window.

Whether it is the original input of the user, or it was suggested by FIX or GUESS, eventually a query that satisfies PASS is sent to the underlying database management system. A non-null result is displayed in the *response* window. A null result may indicate problems of "miscommunication", such as an erroneous presupposition on behalf of the user, or failure to express intentions correctly in a query (it may also be a genuine null result). A mechanism called NULL is then engaged to analyze the situation. It may either detect erroneous presuppositions on behalf of the user, in which case it displays the message "Erroneous presupposition ... cannot answer even simpler queries", or it may decide that the null result is genuine, in which case it displays the message "No data matched ... partial results available". The *query* window shows either the erroneous presuppositions (in the form of queries), or the related queries that have non-null results.

3 The Mechanisms of Flex

FLEX has five major mechanisms: PASS, FIX, GUESS, BROWSE and NULL. Because of space limitations, individual issues and solutions are only sketched here. For more details see [Motro 1986a], [Motro 1986b], [Motro 1986c]. Three mechanisms (FIX, GUESS and BROWSE) make use of an auxiliary database relation called *lexicon*. This two-column relation maps database values onto the attributes in which they appear. All mechanisms consult the schema of the database, which provides important semantic information.

3.1 PASS

The database environment of FLEX is relational, and formal requests are specified with the following statement, reminiscent of SQL's *select* statement [Chamberlin et al 1976]:

retrieve attribute₁, ..., attribute_n from relation₁, ..., relation_m where condition

condition is either of the form *attribute* θ *value* or *attribute*₁ θ *attribute*₂ (where θ is a comparator such as =, \neq , <, >, \leq , \geq), or a combination of such conditions with the logic connectors and, or and not. The result of this query is defined by a Cartesian 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*.

In our examples, we shall assume the following database on musical compositions (key attributes are underlined):

```
composer = name, country, year-of-birth, year-of-death
composition = title, author, type
```

For example, to retrieve the German composers who wrote symphonies, one issues the following query:

```
retrieve name from composer, composition
where country='Germany' and name=author and type='symphony'
```

The PASS mechanism is the simplest component. It checks the syntax of the query and verifies its semantics against the schema of the database. If the query is found to be proper, PASS translates it into the retrieval language of the underlying database management system, and sends it for processing.

3.2 FIX

The FIX mechanism looks for input problems (mostly omissions) that can be corrected unambiguously. For example, the inputs

```
retrieve country from composer where composer='Mozart'
retrieve country where name='Mozart'
retrieve country where 'Mozart'
```

are all improper. All are corrected automatically by FIX to

```
retrieve country from composer where name='Mozart'
```

In the first case a relation name (*composer*) was understood to mean its key attribute (*name*). In the second and third cases the from clause was inferred from the names of the attributes mentioned in the query. In the third case the attribute name was assumed after 'Mozart' was found (in the lexicon) to be a value of that attribute. There are numerous other opportunities for automatic correction.

Consider now the input

```
retrieve composer where '1807'
```

As the value '1807' appears under both *year-of-birth* and *year-of-death*, the user is requested to clarify: "Is '1807' value of *year-of-birth* or *year-of-death*?" Assuming the user meant the former, FIX corrects the input as follows:

```
retrieve name from composer where year-of-birth='1807'
```

Note that FIX also substituted *composer* with its key attribute name and provided the from clause.

3.3 GUESS

If the input is not structured enough to be salvaged by FIX (or if the user cannot answer the disambiguation questions) then GUESS is engaged.

Using knowledge of the functional dependencies among the attributes of the database (inferred from knowledge of the keys), GUESS constructs a permanent semantic network representation of the schema of the database. GUESS then extracts from the input a set of *tokens*. A token is either a *data value* (i.e., appears in a database relation), or a *metadata value* (i.e., appears in the database schema). Each token is then used to *mark* a node in the network. A metadata value marks its own node; a data value marks the node of the attribute under which it appears (again, this information is found in the lexicon). The set of marked nodes is a model for the input. The connection of this set into a subgraph provides an interpretation of the input. Once the connection is made, GUESS infers a default query from the subgraph.

As an example, consider

list the name and the country of the composer of 'The.Magic.Flute'

Obviously, the structure of this input is too far from the formal syntax to be correctable by FIX. Consequently, GUESS is engaged and extracts a total of four tokens. Three metadata values: name, country and composer, and one data value: The.Magic.Flute. The first three tokens mark the data nodes by these names; the last token marks the node title. The subgraph that connects them yields the following query:

```
retrieve name, country from composer, composition
where name=author and title='The.Magic.Flute'
```

The process of inferring a query from a set of tokens involves uncertainties at three phases. First, a token may correspond to more than one node; for example, a data value that appears under more than one attribute. Such *ambiguities* are resolved via a dialogue. Second, there may be several ways to connect the marked nodes. GUESS looks for the most compact subgraph that spans them (this graph-theoretic problem is known as the *Steiner tree problem*). And, third, numerous queries could be inferred from the same connected subgraph (GUESS constructs a default query, preferring conjunctions).

3.4 BROWSE

When the input tokens extracted by GUESS do not include metadata values, it is impossible to construct queries. In this case, FLEX engages BROWSE, requesting it to construct a browsing request for recognized data values.

The primary innovation of this browser, is that it presents each relational database as a single network of objects, making its actual tabular representation transparent. Such networks can support browsing functions of greater utility. The network representation is constructed with the help of the lexicon, which, in effect, "inverts" the database. This enables BROWSE to effect "object behavior": all occurrences of a particular data value throughout the database are considered collectively to be one *object*; this object is *related* to other objects through the functional dependencies in which its individual occurrences participate. Given a data value, BROWSE can construct the appropriate object and its relationships. The effect resembles a semantic network, in which users can browse by mentioning a data value and receive a frame of information on this value.

For example, consider the input

what is known about Mozart

The only recognized token here is Mozart, which is a data value. Consequently, BROWSE is asked to construct a browsing request on Mozart (when several data values are recognized, the user is asked to select one). According to the lexicon the value Mozart appears under both `composer.name` and `composition.author`. In `composer`, Mozart it is a key value, and it is therefore linked to all other objects that occur in its tuple. In `composition`, it is not a key value, and it is therefore linked to the objects that occur as keys in its tuples. Altogether, the objects linked to Mozart create the following frame of information, which is delivered to the user:

Mozart is	name of composer having country	Austria
	name of composer having year-of-birth	1756
	name of composer having year-of-death	1791
	author of composition having title	The_Magic_Flute
	author of composition having title	Jupiter
	author of composition having title	Requiem

At this point the user can continue by entering one of the objects that appear in the frame as the new topic (e.g., Austria). Again, BROWSE will be engaged to construct a new frame of information.

3.5 NULL

Consider a query to retrieve all the operas written by composers from Freedonia. As there are no titles of compositions whose type is opera and whose composer's country is Freedonia, the system returns a null result. This response, however, is misleading. Clearly, the author of this query seems to think that there are composers from Freedonia, while, in fact, there are no such composers; indeed, there isn't even a country Freedonia.

We distinguish between *genuine* nulls, and these *fake* nulls that actually reflect erroneous presuppositions on behalf of the user. Fake nulls are misleading, as they are often mistaken for genuine nulls (and may therefore be understood as reaffirmation of the user's presuppositions). Even genuine nulls 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 (Customer: "How many recordings of Beethoven's 10'th Symphony are available?" Clerk: "Beethoven only wrote 9 symphonies"), and partial answers are usually suggested when the query is legitimate, but does not have an answer (Customer: "Do you have a recording of the Beethoven's 9'th Symphony with Toscanini?" Clerk: "No, but I do have other performances of this piece").

The NULL mechanism attempts to infer the presuppositions of users, test their correctness, and deliver partial results when appropriate. We begin our description of NULL with these observations:

1. Every query reflects a presupposition that the condition it expresses is plausible (may possibly succeed). For example, the query "operas by Mozart" reflects a presupposition "there may be operas by Mozart".
2. Each presupposition is a source of more general (weaker) presuppositions. For example, from the presupposition "there may be operas by Mozart" the presuppositions "there may be operas" and "there are compositions by Mozart" may be inferred.
3. 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 "operas" or "compositions by Mozart" than about "operas by Mozart".

We may summarize this as follows: while users expect that their queries may possibly have null results, they tend to be confident that every more general query would not have failed. Consequently, we adopt the following test: When a query fails, we generate a set of immediate generalizations and attempt them. If all succeed, it is an indication that the original null result was "genuine". The results of the generalizations may then be considered "partial results". If at least one of the immediate generalizations fails, it is an indication that the original null result was "fake". The failed queries (both the original and the generalization) then reflect erroneous presuppositions.

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.

As an example, consider again the previous query to retrieve "titles of operas by composers from Freedonia". Its result is null, so NULL generalizes it to "titles of compositions by composers from Freedonia" and "titles of operas". The result of the first of these queries is still null, so it is generalized to "titles of compositions" and "composers from Freedonia". The result of the second query is still null, so it is generalized to "composers". Its result is non-null. Therefore, the previous query ("composers from Freedonia") reflects a significant erroneous presupposition. The system displays the message "Possible erroneous presupposition ... cannot answer even simpler queries", and the following query is displayed in the query window:

1. retrieve name from composer where country='Freedonia'

As another example, assume the previous query is modified to "titles of operas by composers from Estonia" and that the result of this query is also null. Again, the query is generalized to "titles of compositions by composers from Estonia" and "titles of operas". Here, however, both queries return non-null results. Therefore, the answer to the original query is a genuine null. The system displays the message "No data matched ... partial results available", and the following queries are displayed in the query window:

1. retrieve title from composition where type='opera'
2. retrieve title from composition, composer
where author=name and country='Estonia'

The user can then select any of these queries and see their result in the response window.

4 Conclusion

This paper described the design of FLEX, a tolerant and cooperative user interface to databases. FLEX is tolerant, because it never rejects queries, and it is cooperative, because it never delivers null results without explanation and assistance. Because it "goes to work" only *when* needed and only *as much* as needed, FLEX can be used satisfactorily by experts as well as novices.

As the interface we described is being implemented, research on FLEX is still continuing. One research goal is to incorporate a mechanism for interactive construction of queries. Another research goal is to enable FLEX to cope with "meta queries", queries that are not directed at the data, but at *knowledge* about the data, or about the system itself.

References

- Codd, E. F., Arnold, R. S., Cadiou, J-M., Chang C. L., and Roussopoulos, N. *Rendezvous version 1: an Experimental English Language Query Language System for Casual Users of Relational Databases*. Technical Report RJ2144, IBM, San Jose, California, February 1978.
- Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W., Reisner, P., and Wade, B. W. SEQUEL 2: a unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560-575, November 1976.
- DBASE-III Reference Manual*. Ashton-Tate, Culver City, California, 1984.
- Fogg, D. Lessons from a 'living in a database' graphical query interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 100-106, Boston, Massachusetts, June 18-21, 1984.
- Harris, L. R. Natural language front ends. In *The AI Business*, pages 149-161, The MIT Press, Cambridge, Massachusetts, 1984.
- Herot, C. Spatial management of data. *ACM Transactions on Database Systems*, 5(4):493-513, December 1980.
- Hendrix, G. G., Sacerdoti, E. D., Segalowicz, D., and Slocum, J. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105-147, June 1978.
- INGRES Reference Manual*. Relational Technology, 1984.
- Motro, A. BAROQUE: an exploratory interface to relational databases. *ACM Transactions on Office Information Systems*, 4(2):164-181, April 1986a.
- Motro, A. Constructing queries from tokens. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 120-131, Washington, D. C., May 28-30, 1986b.
- Motro, A. SEAVE: a mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4):312-330, October 1986c.
- McDonald, N., and Stonebraker, M. CUPID: a user friendly graphics query language. In *Proceedings of the ACM-Pacific Conference*, pages 127-131, San Francisco, California, 1975.
- ORACLE User's Guide*. Oracle Corporation, 1983.
- Stonebraker, M., and Kalash, J. TIMBER: a sophisticated database browser. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, pages 1-10, Mexico City, Mexico, September 8-10, 1982.
- UNIFY Reference Manual*. UNIFY Corporation, Lake Oswego, Oregon, 3.0 edition, 1983.
- Wong, H. K. T., and Kuo, I. GUIDE: a graphical user interface for database exploration. In *Proceedings of the Eighth International Conference on Very Large Data Bases*, pages 22-32, Mexico City, Mexico, September 8-10, 1982.