A Trio of Database User Interfaces for Handling Vague Retrieval Requests

Amihai Motro
Computer Science Department
University of Southern California
University Park, Los Angeles, CA 90089-0782

Abstract

We discuss three user interfaces for situations that involve vague retrieval requests: (1) BAROQUE, a browser for relational databases; (2) VAGUE, a query interpreter for relational databases that can handle neighborhood queries (formal queries with *similar-to* comparators); and (3) FLEX, an adaptive interface to relational databases that can service satisfactorily users with different levels of expertise (users who submit queries of different levels of correctness and well-formedness).

1 Introduction

Most database systems and their associated retrieval tools are designed under the assumption that requests for retrieval would be precise and specific. In this paper we discuss three user interfaces, called BAROQUE, VAGUE and FLEX, that were developed for situations that involve *vague* retrieval requests. The term "vague" refers here to two different situations:

- The user is *naive* and cannot express a formal query. This may be either because he is not familiar with the data model used by the system, or with its formal query language, or with the particular database he wishes to access. It may also be because he does not have a specific retrieval goal, or cannot express his goal in the terms required by the system.
- The user is *sophisticated* and possesses all the knowledge required to use the system expertly. However, his request requires a condition which is vague or imprecise.

BAROQUE is a browser for relational databases; VAGUE is a query interpreter for relational databases that can handle neighborhood queries (formal queries with *similar-to* comparators); and FLEX is an adaptive interface to relational databases that can service satisfactorily users with different levels of expertise (users who submit queries of different levels of correctness and well-formedness). Thus, BAROQUE and FLEX are capable of handling vague requests of the first kind, while VAGUE handles vague requests of the second kind.

All three interfaces access standard relational databases that are managed by the INGRES database system [10].

This work was supported in part by NSF Grant No. IRI-8609912 and by an Amoco Foundation Engineering Faculty Grant.

2 BARQOUE

To improve their usability and responsiveness most database systems offer their users a wide variety of interfaces, suitable for different levels of expertise and different types of applications. A particular kind of interface which is now commonly available are *browsers*. Browsers are intended for performing *exploratory searches*, often by *naive users*. Thus, they usually employ simple conceptual models and offer simple, intuitive commands. Ideally, browsing should not require familiarity with the particular database being accessed, or even preconceived retrieval targets. While browsing, users gain insight into the contents and organization of the searched environment. Eventually, the search either terminates successfully or is abandoned.

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. An example of this approach is the interface designed and implemented by Cattell [1]. The interface is to an entity-relationship database, and it features a set of directives for scanning a network of entities and relationships, and presenting each entity, together with its context in a display called *frame*.

Browsers have also been developed for relational systems, for example, SDMS [8], TIMBER [18] and DBASE-III [4]. These are actually tools for scanning relations (including relations that are results of formal 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 value while browsing, and wants to know more about it, he must determine first in what other relations this value may appear (quite difficult), then formulate a formal query, and resume browsing in the new relation. Satisfying questions such as "Is x related in any way to y?" is impossible without an extensive scan of the database.

BAROQUE [13] is a relational user interface, designed to address these deficiencies. With the help of an additional relation, but without affecting the existing relational database otherwise, BAROQUE effectively replaces the record-oriented view which is inherent in the relational model, with a network view, making its actual tabular representation transparent. Such networks can support browsing functions of greater utility. More specifically, BAROQUE employs an "entity directory" relation, which associates every database value with the attributes under which it appears. By thus "inverting" the database, BAROQUE can effect "entity behavior": all occurrences of a particular data value are considered collectively as one entity; this entity is related to other entities through the functional dependencies in which the individual occurrences participate. When the user specifies a data value, BAROQUE can construct the appropriate entity and its relationships. The effect resembles a semantic network , in which users can browse with functions like "Describe x" or "List others like x" or "Explain the connection between x and y". Such *access by value* is especially important for users with no knowledge of the organization of the database. For example, the request "Describe CHAPLIN" would construct a frame named CHAPLIN which tabulates all its relationships to other entities; e.g., NAME of PERSON having: FIRST_NAME CHARLES, COUNTRY BRITAIN, YEAR_OF_BIRTH 1889,... DIRECTOR of FILM having TITLE: MODERN TIMES, THE GOLD RUSH, THE CIRCUS,...

Among its other features, the system incorporates the database schema into the same representation; it allows users to switch rapidly back and forth between formal querying (in QUEL [19])

and browsing; and it relies on menus to facilitate communication with users. Upon entry to browse mode, the name of the database is established as the default topic. Thus, this most general entity is provided to the user as the end of a thread. Following it the user may survey the database, and ultimately reach every other entity.

The cost entailed by the browser, in terms of the additional space to store the entity directory 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 sufficient storage is unavailable, it is possible to implement only part of the entity network, by inverting on selected attributes only. All other values will be listed while browsing in their neighborhoods, but they may not become topics of browsing requests. This has the interesting effect of distinguishing between actual *entities* that participate in relationships, and simple *properties* that describe entities. In fact, the resulting model resembles the Entity-Relationship approach to data modeling. One possible strategy for selective inversion is to invert only on attributes that are keys or foreign keys. Under this strategy, every entity that is assembled occurs at least once as the value of the key in some relation.

Our method for assembling entities is based only on identities of data values. Consequently, values that possess different meaning altogether, but are expressed with the same string of characters, are assembled into one entity. This weakness can be attributed to the limited semantic capabilities of the basic relational model, where the only information available on the meanings of the different attributes are their names and their primitive types (e.g., integer, character). A well-known enhancement to the relational model uses a stronger concept of *domains* to classify the attributes. This enhancement can be readily incorporated into BAROQUE to assemble separate entities for values that belong to multiple domains. For example, assume a database with attribute SALARY defined over the domain DOLLARS, and attribute YEAR_OF_BIRTH defined over the domain YEARS. If the value 1889 appears under both PRICE and YEAR_OF_BIRTH, BAROQUE will create two separate entities: 1889 DOLLARS and 1889 YEARS. Notice, however, that even with the current approach, the names of relationships in which 1889 participates provide different *interpretations* for this entity. For example, BAROQUE will describe the topic 1889 with both PRICE of ITEM having ITEM_NO 6710 and YEAR_OF_BIRTH of PERSON having NAME CHAPLIN. Thus, while the information included in these answers combines different semantics of the entity 1889, it is interpreted clearly, and the user can disregard the portion of the answer that is irrelevant.

3 VAGUE

Requests for data can be classified roughly into two kinds: *specific* queries and *vague* queries. A specific query establishes a rigid qualification, and is concerned only with data that match it precisely. Some examples of specific queries are "How much does Jones earn?" or "When does flight 909 depart?" If the database does not contain salary information on Jones or departure time for flight 909, null answers should be returned; the user is not interested in the earnings of somebody else or in the departure time of a different flight. A vague query, on the other hand, establishes a target qualification and is concerned with data that are *close* to this target. As an example, consider "List the inexpensive French restaurants in Westwood". If there are none, a moderately priced Continental restaurant in Santa Monica may have to do. Similarly, when a project calls for experienced C programmers with background in applied mathematics, we may want the personnel database to mention also that there is an engineer with some knowledge of Pascal.

While many retrieval requests are intrinsically vague, most conventional database systems cannot handle vague queries directly. Consequently, they must be emulated with specific queries. Usually, this means that the user is forced to retry a particular query repeatedly with alternative values, until it matches data that are satisfactory. If the user is not aware of any close alternatives, then even this solution is infeasible. A system that allows users to express vague queries directly is more cooperative, and possibly more efficient. While issues of vague retrieval have been addressed in related disciplines, particularly information retrieval (see [17] or [20]) and fuzzy systems (e.g., [16, 22]), little has been done to extend current relational database technology to provide adequate tools for performing vague retrieval (one exception is [9]). VAGUE [15] is a system that extends the relational data model to provide it with vague retrieval capabilities.

To determine similarity between data values we introduce the notion of distance. Each database domain is provided with a definition of distance between its values, called *data metric*. For example, in a database on restaurants there may be metrics to measure distances between cuisines, between locations, between price ranges, as well as a metric to measure distances between restaurants.

To express vague queries we introduce a vague selection comparator, called *similar-to*. A *similar-to* comparison is satisfied with data values that are within a predefined distance of the specified value. For example, the vague comparison "location *similar-to* Westwood" may be satisfied by Westwood, Santa Monica and Beverley Hills.

Thus, the previous specific query "List the restaurants whose cuisine is French, whose price range is inexpensive, and whose location is Downtown" may be relaxed into a vague query such as: "List the restaurants whose cuisine is *similar-to* French, whose price range is *similar-to* inexpensive, and whose location is *similar-to* Downtown".

This model is quite straightforward, and its satisfactory operation relies almost entirely on the quality of the metrics that are provided for the individual domains. Here, VAGUE allows the database designer four choices. He could use one of several *built-in* metrics; he could provide a procedure that *computes* the distance between every two elements of the domain; he could provide a relation that *stores* the distance between every two elements of the domain; or he could use a *reference* relation (an existing database relation that is keyed on this domain). In the latter case, distances between elements of the domain would be defined as distances between their tuples in the reference relation, where tuple distance is defined as a combination of the individual distances between their corresponding components.

Each tuple in the answer to a query that includes several vague qualifications involves several deviations from the specific values mentioned in these qualifications. By combining these individual deviations into a single value, VAGUE can present the answer to the user in order of optimality. Thus, there are two occasions where VAGUE combines several component distances into a single distance: in one of its metric types, and in the presentation of vague answers.

The design of VAGUE reflects two fundamental requirements: conceptual simplicity within a relational framework and adaptability.

The purpose of VAGUE is to enhance a relational database system with vague retrieval capabilities. An important design guideline is to realize this goal with only minimal deviation from this popular model. The relational data model is extended with a single concept: *data metrics*, and the query language is extended with a single feature: a *similar-to* comparator. (Indeed, the relational data model is *generalized*, since a non-metricized database is a particular type of a metricized

database.) To present queries, users need only to know about the new comparator.

To be useful, a system that implements vague queries, must be able to adapt itself to the views and priorities of its individual users. VAGUE incorporates three adaptability features. (1) Often, distances between values of a given domain may be measured according to various metrics. For example, distances between values of domain CITY may be defined in miles "as the crow flies", or as shortest driving distances, or even as differences between the names of the cities. VAGUE permits *multiple metrics for the same domain*. When a query makes use of a *similar-to* comparator, the user is presented with the various possible semantics of this comparator in its present context, and is asked to select. (2) With referential metrics, one of the metric types available in VAGUE, individual users are allowed to *influence the definition of the metric* according to their own views. For example, the distance between two cities may be defined as a combination of the distances between some of their available attributes, such as size of population, climate, and employment rate. If such a metric is selected, the user is allowed to judge the relative importance of the various attributes in the overall distance. (3) When a query involves several vague qualifications, users are allowed to *express their relative importance* in the overall query. For example, consider the previous vague query about restaurants whose cuisine is similar to French, whose price range is similar to inexpensive, and whose location is similar to Downtown. Each tuple in its answer involves three deviations from the specified values, which are then combined so that the answer may be presented in order of optimality. However, it may be that the user has different willingness to compromise on the various qualifications; for example, the user may be willing to compromise more on the type of the restaurant than on its price range or location. VAGUE allows users to express their relative willingness to compromise, and uses this input in the definition of the corresponding metric.

The design of VAGUE represents a compromise between the sometimes conflicting requirements for simplicity, flexibility and efficiency. Some examples of design compromises are described below. Users of VAGUE cannot provide their own similarity thresholds for each vague qualification. It was observed that this will require that users become familiar with particular data metrics. Instead, VAGUE allows its users to double the threshold and repeat the query. Similarly, except for the ability to enter weights for referential metrics, users of VAGUE are limited to interpretations of similarity (i.e., metrics and thresholds) that have been provided by others. While it is possible to design an interface that will permit users to introduce their own interpretations of similarity, it was determined that the complexity of this task usually would exceed the expertise of many users, especially casual users. Instead, this task is reserved for database designers or administrators, and users are invited to select from menus of metrics that are currently supported. To prevent the querying process from becoming too tedious to the user, VAGUE tries to be economical in its dialogue with the user. At several places it may be possible to gain flexibility by additional interaction; for example, when a vague query does not match any data, it is possible to ask the user which *similar-to* comparator should be weakened (currently, all are weakened simultaneously). Finally, since each tuple in an answer to a vague query must satisfy *all* the vague qualifications, it is possible that a tuple would not be retrieved, even if its total compromise is smaller than that of tuples that were retrieved. This approach was adopted primarily for reasons of efficiency. In addition, because the combination of individual distances into a single distance is sometimes risky, VAGUE prefers not to rely on it for *determining* its answers, only for *ranking* them.

The issue of appropriate similarity measures for retrieval has been researched and debated extensively. Our purpose in designing and implementing VAGUE is not to resolve this issue by adopting any one particular approach, but to provide relational databases with a flexible mechanism, with which different kinds of data metrics may be implemented and tested.

A legitimate concern is that vague queries will be satisfied by meaningless values. Careful selection of the metrics and the parameters during the design of the database is extremely important. Also, by extending the answers to include the values with which the vague qualifications were satisfied (a feature available in VAGUE), users can monitor the judgements made by the system. Finally, it can be assumed that users who consciously present vague queries (to systems or to humans) are well aware of the fact that subjective judgement is involved, and would probably examine answers to vague queries more carefully than answers to specific queries.

4 FLEX

A common method for accessing databases is via query language interfaces (e.g., QUEL [19], SQL[2]). 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 (e.g., QBF [10]), graphical interfaces (e.g., CUPID [12], GUIDE [21], LID [5], SKI [11]), natural and pseudo natural language interfaces (e.g., RENDEZVOUS [3], LADDER [7], INTELLECT [6]), and browsers (e.g., TIMBER [18], SDMS [8], BAROQUE [13]). These interfaces are oriented towards non-programmers, and therefore require only limited computer sophistication. Expressing requests may be as simple as selecting from a menu or 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 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.

FLEX [14] is an experimental user interface designed to be used satisfactorily by users with different levels of expertise. It 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. FLEX is also *cooperative*. It never delivers null answers without explanation or assistance. This tolerant and cooperative behavior is modeled after human behavior, and is thus reminiscent of natural language interfaces.

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 user input is *cascaded* through this series of mechanisms, until an interpretation is found.

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 transformations involve both syntactic and semantic corrections, as well as synonym substitution. 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 tokens that are recognized in the input. The basic approach of the synthesizer is to model the entire database as a graph, mark the nodes that correspond to tokens that are recognized in the input, span these nodes with a minimal tree, and then translate the tree into a formal query. 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 display frames of information extracted from the database on the recognized input tokens. The basic approach of this mechanism is essentially similar to that of BAROQUE.

Hence, FLEX never rejects queries, and the accuracy and specificity of its interpretations correspond to the correctness and well-formedness of the input.

Because it is engaged only *when* needed and only *as much as* needed, FLEX can be used satisfactorily by users with different levels of expertise, and thus appeal to a more universal community of users. For example, a perfect formal query submitted by an expert will be executed immediately without any modification; while a single word submitted by a novice will flow through the entire sequence of mechanisms until finally it will result in a frame of information about this word. FLEX may be viewed as an interface that *adapts* to the level of correctness and well-formedness of its input (providing interpretations of corresponding accuracy and specificity).

This ability to adapt is complemented with features of *cooperative* behavior, whereby null answers are never delivered without explanation or assistance. If the final answer is null, the original query is passed to a query *generalizer*, which issues a set of more general queries to determine whether the null answer is *genuine* (it then suggests related queries that have non-null answers), or whether it reflects *erroneous presuppositions* on behalf of the user (it then explains them). The basic heuristic applied here is that a null answer is genuine, if and only if all the queries that are more general have non-null answers.

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.

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.

The "knowledge base" used by FLEX consists of three auxiliary relations, that are stored along with the database itself: a DICTIONARY that stores the definition of the database, a LEXICON that maps database values to the attributes under which they appear, and a THESAURUS, which stores synonyms of recognized database tokens. 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, 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 (a similar relation was used in BAROQUE). 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.

Work on FLEX is continuing. Current goals include extension of the retrieval language to include a fuller set of operators, improved performance of the generalizer, and improved presentation.

5 Conclusion

FLEX represents our current approach regarding user interfaces to databases, that prefers a single interface with universal appeal over a cluster of specific tools. Thus, we would prefer to incorporate tools like BAROQUE and VAGUE into FLEX. Indeed, some of BAROQUE's capabilities are already available in the last mechanism of FLEX. As VAGUE is a formal extension of the relational database model, it is, indeed, "orthogonal" to FLEX, and both could be combined without conceptual difficulties. In other words, FLEX should become an interface to VAGUE.

References

[1] R. G. G. Cattell. An entity-based database interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Santa Monica, California, May 14–16), pages 144–150, ACM, New York, New York, 1980.

[2] 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 Journal of Research and Development*, 20(6):560–575, November 1976.

[3] E. F. Codd, R. S. Arnold, J-M. Cadiou, C. L. Chang, and N. Roussopoulos. *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.

[4] *DBASE-III Reference Manual.* Ashton-Tate, Culver City, California, 1984.

[5] D. Fogg. Lessons from a 'living in a database' graphical query interface. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Boston, Massachusetts, June 18–21), pages 100–106, ACM, New York, New York, 1984.

[6] L. R. Harris. Natural language front ends. In *The AI Business*, pages 149–161, The MIT Press, Cambridge, Massachusetts, 1984.

[7] G. G. Hendrix, E.D. Sacerdoti, D. Segalowicz, and J. Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147, June 1978.

[8] C. Herot. Spatial management of data. *ACM Transactions on Database Systems*, 5(4):493–513, December 1980.

[9] T. Ichikawa and M. Hirakawa. ARES: a relational database with the capability of performing flexible interpretation of queries. *IEEE Transactions on Software Engineering*, SE-12(5):624–634, May 1986.

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

[11] R. King. Sembase: a semantic dbms. In *Proceedings of the First International Workshop on Expert Database Systems* (Kiawah Island, South Carolina, October 24–27), pages 151–171, Institute of Information Management, Technology and Policy, University of South Carolina, Columbia, South Carolina, 1984.

[12] N. McDonald and M. Stonebraker. Cupid: a user friendly graphics query language. In *Proceedings of the ACM-Pacific Conference* (San Francisco, California), pages 127–131, ACM, New York, New York, 1975.

[13] A. Motro. BAROQUE: a browser for relational databases. *ACM Transactions on Office Information Systems*, 4(2):164–181, April 1986.

[14] A. Motro. FLEX: a tolerant and cooperative user interface to databases. *IEEE Transactions on Knowledge and Data Engineering*, to appear.

[15] A. Motro. VAGUE: a user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3):187–214, July 1988.

[16] H. Prade and C. Testemale. Generalizing database relational algebra for the treatment of incomplete or uncertain information and vague queries. *Information Sciences*, 34(2):115–143, November 1984.

[17] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, New York, 1983.

[18] M. Stonebraker and J. Kalash. TIMBER: a sophisticated database browser. In *Proceedings of the Eighth International Conference on Very Large Data Bases* (Mexico City, Mexico, September 8–10), pages 1–10, VLDB Endowment (available from Morgan-Kaufmann, Los Altos, California), 1982.

[19] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[20] C. J. van Rijsbergen. *Information Retrieval (Second Edition).* Butterworths, London, 1979.

[21] H. K. T. Wong and I. Kuo. GUIDE: a graphical user interface for database exploration. In *Proceedings of the Eighth International Conference on Very Large Data Bases* (Mexico City, Mexico, September 8–10), pages 22–32, VLDB Endowment (available from Morgan-Kaufmann, Los Altos, California, 1982.

[22] M. Zemankova and A. Kandel. Implementing imprecision in information systems. *Information Sciences*, 37(1,2,3):107–141, December 1985.