ENCYCLOPEDIA OF LIBRARY AND INFORMATION SCIENCE

Executive Editor

ALLEN KENT

SCHOOL OF INFORMATION SCIENCES
UNIVERSITY OF PITTSBURGH
PITTSBURGH, PENNSYVANIA

Administrative Editor

CAROLYN M. HALL

ARLINGTON, TEXAS

MARCEL DEKKER, INC. NEW YORK • BASEL

www.dekker.com

Copyright © 2000 by Marcel Dekker, Inc.

COOPERATIVE DATABASE SYSTEMS

Introduction

Database systems are computerized systems for managing large amounts of data. Their principal function is to process requests from *users* to retrieve or alter the data they store. Traditionally, such requests must be expressed in a formal *query language*. Requests that are well defined are performed promptly; other requests are declined.

Processing requests for information is also a frequent human activity, but in contrast with the rather rigid behavior of database systems, humans are capable of *cooperation*. A cooperative person will attempt to guess the information *need* of the requester, even when the request itself is flawed or unsatisfiable, and will then respond to this need rather than the request. Cooperation is considered to be an aspect of human intelligence.

As the term suggests, the area of *cooperative database systems* is concerned with techniques that enhance database management systems with such cooperative behavior. These techniques usually imitate some aspect of human behavior.

This review of cooperative database systems focuses on methods that imitate some aspects of human cooperation, and we note that there are many other methods that enhance the usability and responsiveness of database systems but fall outside the scope of this review. These include graphical user interfaces and query-by-example interfaces, the use of profiles to store user preferences, the extension of query languages to allow users to express vague ("fuzzy") requests, and in general, any method that decreases the skills that users must have. Often these methods are combined with cooperative methods to create database systems that are oriented toward naive users.

Research in cooperative database systems is strongly indebted to research in artificial intelligence, in particular the work done by Joshi, Webber, and others at the University of Pennsylvania (*1–5*). The main interest of that group was natural language discourse between users and systems, with emphasis on mutual beliefs, misconceptions, and cooperation. Database researchers for the most part have deemphasized issues related to natural language understanding, and have concentrated on developing techniques for well-known data models, such as those based on relations or on mathematical logic.

Our review of cooperative database systems is organized around a simple four-category classification of the various types of cooperative behavior. (See the section "Classification" below.) This classification is based on the stage in the user–system interaction process in which cooperation is introduced. The sections "Formulation of Objectives and Conversion of Objectives to Queries," "Detecting Deficiencies in Apparently Good Queries," and "Annotation and Explanation" are dedicated to these four categories, and the section "Assuring that Queries Correspond to Objectives" discusses a related issue. In the section "Prototype Systems" we briefly describe three systems that combine different cooperative techniques in a single system. The Conclusion ends this article with a discussion of problems that remain to be solved. We begin with a quick review of the formal concepts that will be used throughout this article.

Background

Most of the work in the area of cooperative databases, in particular the work done in the last ten to fifteen years, has been in the framework of the relational data model (*6*, pp. 43–65) and the more general logic-based data model (*6*, pp. 96–106), yet most of the techniques described may be ported to other data models. The basic structures of the relational or logic models are described below.

A *relation scheme* R is a finite set of *attributes* $A_1,\ldots,A_m$. Associated with each attribute A_i is a set of values called the *domain* of A_i. A *tuple* t of relation scheme R is a sequence of values $(t_1,\ldots,t_m)$, where t_i is an element of the domain of A_i. A relation instance r on the relation scheme R, denoted $r(R)$, is a finite set of tuples of R. A *database scheme* D is a set of relation schemes $R_1,\ldots,R_n$. A database instance d of the database scheme D, denoted $d(D)$, is a set of relation instances $r_1(R_1),\ldots,r_n(R_n)$. The logic data model simply regards each relation scheme R as the definition of a predicate with m variables, and each relation instance $r(R)$ as an extension of that predicate.

A *constraint* c on the database scheme D is a closed formula in predicate logic whose predicates are from the scheme D. A constraint c is *satisfied* by a database instance $d(D)$ if the interpretation of c in the instance d is true. Constraints are used by database systems to control the integrity of their data. This is achieved by refusing any changes to the present instance that would result in an instance that would violate any of the constraints. By themselves, integrity constraints are an important source of semantic information (knowledge), as they express real-world relationships among the data items.

Finally, *a query* Q on the database scheme D is a formula in predicate logic whose predicates are from the scheme D. The *answer to* Q in a database instance $d(D)$ is a set of tuples; each tuple is a combination of values from d whose substitution for the free variables of Q makes Q true.

As an example, consider the following database scheme:

Employee(Name,Sex,Department,Title,Salary)
Department(Dname,Telephone,Budget)

An example of an instance of this scheme is

Employee: (john,male,design,engineer,52000)
(betty,female,support,programmer,45000)
(harry,male,support,receptionist,30000)
Department: (design,x1665,600000)
(support,x2144,100000)

Two examples of constraints are

$$(\forall x,y,z,u,v)\ Employee(x,y,z,u,v) \rightarrow (\exists w)\ Department(u,w)$$
$$(\forall x,y,z,v)\ Employee(x,y,z,\text{engineer},v) \rightarrow v > 50000$$

The first constraint guarantees that only departments that occur in the *Department* instance are used in the *Employee* instance. (Such constraints are usually called *referential* constraints.) The second constraint states that engineers earn salaries of at least $50,000.

The following query retrieves the names of the female engineers who earn over $60,000 and the telephone numbers of their departments. (The free variables in this query are x and u.)

$$(\exists y,z,v)\ Employee(x,\text{female},y,\text{Engineer},z) \wedge Department(y,u,v) \wedge z > 60000$$

In practice, queries are often written using an alternative notation called relational algebra. In commercial database systems, queries are written in notations that resemble a programming language, notably SQL.

Classification

Our classification of cooperative database techniques is based on the stage of the retrieval process at which cooperation is introduced. A typical database interaction is an iterative process, as follows:

repeat
  (Re)formulate *objective*
  Convert *objective to query*
  Submit *query*
  Examine *answer*
until *objective* is satisfied

In this process, a user approaches the database system with a specific data requirement, termed the *objective*. The user then expresses this objective in a formal *query*, which is submitted to the database system. Upon examining its result, the user determines whether the objective has been satisfied or not. In the former case, the process terminates; in the latter case, it is repeated; the user may now modify the objective, or he may convert the same objective to a different query (i.e., correct the query). To allow users to abandon the querying process, a special objective *null* may be assumed, which is always converted to an empty query and is vacuously satisfied.

This typical interaction suggests a simple classification of the kinds of cooperation techniques that database systems may offer.

1. *Formulation of objectives.* Techniques in the first category address situations in which a database user is essentially saying "I don't know what I want"; that is, the user does not have an objective or the objective is not specific enough. The purpose of the cooperation is to help the user understand his needs and formulate his objective.
2. *Conversion of objectives to queries.* Techniques in the second category address situations in which a user is essentially saying "I know what I want, but I don't know how to say it"; that is, the user has a specific objective but is incapable of expressing it in the formal language that is supported by the database system. The purpose of the cooperation is to formulate a query that corresponds to the objective and is acceptable by the database system.
3. *Detecting deficiencies in apparently good queries.* Now assuming that users know what they want and how to correctly state it in good queries, the third category includes techniques that analyze the submitted queries to detect deficiencies in the objectives that they express.* In other words, the system is essentially saying to the user "Your query is technically correct, but there is doubt that this is what you really want."

*At this point a query is assumed to correspond correctly to an objective, so one could talk about either the objective or the query as deficient.

4. *Suggestion of additional information and explanation*. In this final category the query was found to be acceptable from all aspects and was executed without any problem, yet the system has reason to believe that it has some additional information that the user would appreciate, even though this information was not requested in the actual query.

It is important to note that even the most cooperative database system has its limitations. In general, it is still possible that a query submitted to a database system would be acceptable to the system both syntactically and semantically and would not be suspected for any other deficiency even though the query might not correctly implement the intent (objective) of the user. In such situations the user would be alerted to the problem only if he is familiar with some aspects of the answer and recognizes that the answer that was computed does not satisfy all of these aspects; otherwise such errors go unnoticed. The author is not aware of any system that assists users in detecting such errors. Further discussion of this problem is deferred until the section "Assuring that Queries Correspond to Objectives."

In the next four sections we discuss each of these four categories in detail, with particular emphasis on the latter two categories.

Formulation of Objectives and Conversion of Objectives to Queries

We begin by discussing cooperation techniques that assist users who approach a database system without specific retrieval objectives. The main tool available to such users is *browsing*.

These days the term browsing is identified with searching the World Wide Web (WWW),* yet browsing had been used much earlier as a technique for searching databases. It has long been recognized that there are several situations in which users may benefit from flexible tools that allow them to examine and explore databases. One such situation is when users do not have firm retrieval objectives; browsing the data may help these users formulate such objectives. Another situation that benefits from browsing is when the information provided in the database scheme (e.g., the names of the attributes and their data types) is insufficient for understanding the contents of the database; a brief browsing session might then provide the necessary semantics. Browsing is also beneficial for users who are not well versed in any of the formal languages for querying the database because it usually requires simpler skills.

Basic browsers (often found in early systems) may simply allow users to view the stored data in some order; for example, a browser for a relational database system may allow users to "scroll" each of its relations. More sophisticated browsers employ a conceptual model that interleaves the data in a *network* of some kind, and browsing is done by *navigation*; the user begins at an arbitrary point in the data network (perhaps a standard initial position), examines the data in its "neighborhood," and then issues a new command to proceed in a new direction. These networks will often incorporate

*The WWW is a very large collection of cross-referenced documents that are located on many different computers that are connected via the Internet. Browsing the WWW is an activity that involves the examination of related documents by following the cross-reference links.

the meta data (i.e., the database scheme) as well. In the network structure assumed by most browsers, *adjacency* reflects *relationship*. For example, JOHN is linked to MANUFACTURING because John works in the manufacturing department, and MANUFACTURING is linked to DEPARTMENT because the former is an instance of the latter. An alternative model is to replace the network by a *multidimensional space*. A measure is defined that calculates the similarity among the different data objects, and the objects are places in this space so that *proximity* reflects *similarity*. (Examples of database browsers are given in Refs. *7–18*.)

While browsers may help users formulate specific retrieval objectives, arguably they should not be classified as cooperative tools because they are usually "passive" tools that do not initiate any cooperation. A notable exception is a technique that monitors the activity of users in browsing sessions in an attempt to detect repetitive actions that could then be captured in concise queries that retrieve the same information more efficiently (*19*). The automatic inference of formal queries from information obtained in browsing sessions is also the purpose of the query-by-browsing system (*20*). Users of query-by-browsing are presented with listings of database records and mark them as relevant or irrelevant to their retrieval objectives. The system then infers the pattern underlying the selections by means of an inductive learning-based algorithm. Both techniques go beyond the derivation of search objectives and also attempt to derive queries that carry out these objectives (the subject of the next section).

Presumably users without specific retrieval objectives could also benefit from a system capable of conducting *clarification dialogues*, but apparently this technique, which has been applied in natural language understanding systems, has not become popular in query systems.

The next stage in the prototypical querying process presented in the section "Classification" is to convert retrieval objectives to proper queries in a formal language supported by the system. The main task here for a cooperative system is to help inexperienced users overcome their lack of proficiency in the query language. This task is usually achieved by *interactive query construction*.

The simplest form of interactive query construction is a graphical user interface (GUI) that breaks down the task of constructing queries into sequences of small steps, and that through the use of context-sensitive menus helps users conquer individual steps. Such software interfaces are often termed *wizards*. Still, most wizards require some level of proficiency in the formal language.

A somewhat different construction paradigm was implemented in an early system called RABBIT (*21*). This system applies a paradigm of repetitive reformulation of an initial objective. At each iteration the user is presented with the answer to the current query. Having observed the answer, the user can then refine the query in one of several possible ways.

Detecting Deficiencies in Apparently Good Queries

At this point we may assume that users know what they want and how to correctly state it in queries. Submitted queries are therefore now guaranteed to pass the syntactical and semantical analyses of standard database systems. Even these seem-

ingly proper queries may incorporate various deficiencies in the objectives that they express, however, and we now focus on methods that analyze queries to detect such deficiencies and to suggest possible improvements. We shall discuss these methods in the following four groups:

1. Detection and simplification of objectives/queries that are unnecessarily *complex*.
2. Detection and modification of objectives/queries that are *unsatisfiable*.
3. Detection and modification of objectives/queries that are *incomplete*.
4. Annotation and explanation of answers.

Users who access databases have various *presuppositions* on the meaning of the data and its overall properties. These presuppositions are at the basis of the users' retrieval objectives and are therefore reflected in the queries that are eventually submitted to the database system. In many of the methods to be described, the database system will conclude that there is reason to believe that some of these presuppositions are incorrect. The detection of these *erroneous presuppositions* (we shall also use the term *misconceptions*) is an important manifestation of cooperative behavior.

COMPLEX QUERIES

At times queries are more complex than they need to be. Usually such queries incorporate information that is *redundant* (i.e., satisfied tautologically). This redundancy is often the result of misconceptions on behalf of the users who submitted the queries. By detecting and reporting such redundancies, the database system is informing the users of possible misconceptions. For example, assume a database that has a constraint that requires all professors to have a Ph.D. A user who inquires about "professors with a Ph.D." appears to hold the misconception that some professors might not have a Ph.D. By detecting this redundancy and informing the user of a possible misconception, the database system is demonstrating cooperative behavior. Moreover, by ignoring this redundancy and answering the query as presented, the database system might contribute to the reinforcement of such misconceptions.

The initial motivation for simplifying unnecessarily complex queries was *optimization*: the opportunity to reduce the work of processing the query in the database. As an example, consider the personnel database that was defined in the section "Background," and assume that it enforces the constraint that the starting salary of engineers is $50,000. Consider now a query on the female engineers who earn more than $45,000. Naively, this query requires that every employee record be checked for three qualifications: *Sex*="female," *Title*="engineer," and *Salary*>45,000. When the constraint is considered, however, the last qualification becomes redundant, suggesting an improvement to the naive query processing strategy.

Most research efforts here have been in the framework of the logic data model, and the typical simplification detects redundant conjuncts: a constraint $\alpha \rightarrow \beta$ transforms the query $\alpha \wedge \beta$ to the simpler query α.

This very process can also be motivated as a cooperative procedure (*22–24*). A query with redundancies indicates that the inquirer may have misconceptions regard-

ing the semantics of the environment modeled by the database. A cooperative answer to the previous query would then be a list of female engineers, accompanied by a statement to the effect that *all* engineers earn more than $50,000.

UNSATISFIABLE QUERIES

A query is *unsatisfiable* when it would return an empty answer in *every* database instance; that is, the query would *never* have an answer. Obviously, such queries reflect deficient retrieval objectives. Lack of satisfaction would happen if the query *conflicts* with the stored knowledge (the constraints). In such cases the user is clearly not aware of some important aspect of the environment being modeled by the database. Detecting these conflicts is possible through the machinery of logic. Clearly, by extracting the erroneous presuppositions from the query and alerting the user to their existence, the database system exhibits cooperative behavior.

A related situation is one in which a query returns an empty answer but does not conflict with the stored knowledge. Our conclusion would be only that the current instance of the database contains no data that satisfy the specification expressed in the query. It is still possible that the query is indeed unsatisfiable, however, but the stored knowledge is simply incomplete. In cases in which the query is only *suspect* of being unsatisfiable, various heuristics have been developed to detect possible erroneous presuppositions that might be pertinent. (This situation has also been referred to as a *state* misconception.) This technique is important for database systems that do not use constraints. We begin our discussion with techniques for queries that return empty answers.

Empty Answers

From the various situations in which cooperative behavior is appropriate, probably the one situation that has received the most attention is when database systems return empty answers in response to user queries (*22, 5, 25–28*).

Frequently queries that have empty answers indicate *erroneous presuppositions* of the inquirers. In human conversation, they are often answered with an explanation that points out the error. Standard database systems do not detect these problems, and process such queries in the usual manner. Consider this query about the students who failed the course CS-345 in the summer semester of 1998. Most probably the inquirer assumes that CS-345 was offered in the summer of 1998 and would interpret an empty answer as if all the students passed this course successfully. In reality, it could be that CS-345 was never offered in the summer of 1998. Moreover, that the system did not complain about the query might be interpreted by the inquirer as an *affirmation* that indeed this particular course was offered in that particular period. A more formal presentation of these observations follows, based on Ref. *26*.

Queries often reveal the *presuppositions* of the inquirers. Specifically, a person who requests data believes that such data may exist. (Otherwise, why ask?) This belief becomes even *stronger* when the query is *broadened*; that is, its predicate is replaced by a subsuming predicate. Consequently it is reasonable to assume that users expect that

answers to their queries could possibly be empty, but they do not expect answers to even broader queries to be empty.

In other words, a query that has an empty answer, but all its broader queries have nonempty answers, is within the expectations of the user; we call it a *genuine* empty answer. A broader query that has an empty answer indicates an *erroneous presupposition* of the user, however, the presupposition being simply that data that satisfy the broader query must exist.

In the above example, three broader queries should be tried.

1. The students who received a grade (not necessarily *F*) in CS-345 in summer 1998
2. The students who failed CS-345 (not necessarily in summer 1998)
3. The students who failed a course (not necessarily CS-345) in summer 1998

If all three answers are nonempty, we conclude that the original empty answer is genuine (i.e., in general, the original query is satisfiable). Any empty answer, however, indicates an erroneous presupposition. The corresponding erroneous presuppositions are

1. CS-345 was not offered in summer 1998.
2. The grade *F* was never used in CS-345.
3. The grade *F* was never used in summer 1998.

Assume now that the first of these answers is found to be empty. The same procedure could now be applied to this answer as well. Its broader queries are: (1) students who received a grade in CS-345, and (2) students who received a grade in summer 1998. The corresponding erroneous presuppositions are: (1) CS-345 has never been offered (possibly there is no such course designation), and (2) no courses were held in summer 1998 (possibly there is no such semester designation).

Obviously each of these new erroneous presuppositions is more *significant* than the earlier ones, which leads us to define a *maximal erroneous presupposition* (*MEP*) as one that corresponds to a query that has an empty answer yet all its broader queries have nonempty answers, and to conclude that only MEP are significant. The following procedure summarizes this technique:

1. When a query returns an empty answer, search for all MEP.
2. If none is found,* then the empty answer is genuine.
3. Otherwise, report every MEP to the user.

The specific techniques for gradual broadening of queries depend on the data model. An important aspect of every implementation is how to manage the additional queries efficiently.

*The query itself is the only MEP.

Unsatisfiable Queries

Our interpretation of an empty answer was that a presupposition (that there are data that match the query) was refuted. In effect, the database system is saying that *there are no data* to support this presupposition. In the presence of additional database knowledge (e.g., integrity constraints), it may be possible to conclude that *there can be no data* to support a given presupposition. Such rejection is much stronger and more informative (*26*).

As an example, assume that the previous personnel database enforces two constraints: (1) the starting salary of engineers is at least $50,000, and (2) senior managers earn more than engineers. Consider now a query on the senior managers that earn less than $40,000. A strict answer would be that there are none. A cooperative answer would be that *there can be none*, because all senior managers must earn at least $50,000.

The discovery of unsatisfiable queries is often done together with the discovery of redundancies in queries, a subject that was discussed in the subsection "Complex Queries." As mentioned, these techniques are usually within the framework of the logic database model (*29, 23, 24*) and are also motivated by query optimization: to avoid costly processing of a query in the actual database if it can be established a priori that the query is unsatisfiable.

INCOMPLETE QUERIES

A query is considered *incomplete* when it is highly possible that its "literal" answer would not completely satisfy the user who submitted it, and would therefore trigger follow-up queries. In human interaction, literal answers could be annoying at times, and a cooperative human is expected to anticipate follow-up queries voluntarily. An example of an incomplete query is one that generates an empty answer; in many cases such a query would be followed up by other queries that will "weaken" the specification, with the intention of matching some data. Another example is a query that requests only part of a "cluster" of information that consists of a meaningful whole; quite often such a query would be followed up by additional queries that will ask for the "rest." Techniques in the third category attempt to detect such queries and enlarge them automatically. We will now discuss two techniques that anticipate follow-up queries.

Partial Answers

In the subsection "Empty Answers" we discussed queries that are unanswerable because their answers are empty, and we distinguished between empty answers that are genuine and empty answers that reflect erroneous presuppositions. In both cases broader queries had to be evaluated. The empty answer was declared genuine only if all broader queries had nonempty answers, and a broader query whose answer was also empty indicated an erroneous presupposition.

While a genuine empty answer confirms that the query was "reasonable," very

frequently the user who submitted the query will follow up with one or more queries that relax some aspect of the original query specification with the purpose of retrieving data that fulfill much of the original specification.

The broader queries that were already computed in response to the original query provide precisely this information, thus the techniques discussed in "Empty Answers" have a dual purpose: when an empty answer is declared genuine because all broader queries evaluated to nonempty answers, these nonempty answers are offered in partial fulfillment of the original query. The cooperative trait here can be summarized as "never say none."

As an example, consider a query on the female employees with seven years experience in management, and assume that the strict answer is "none." A cooperative response would offer the user options such as the following:

1. Female employees with seven years experience
2. Female employees with experience in management
3. Employees with seven years experience in management
4. Female employees with six years experience in management
5. Female employees with seven years experience in human resources

The methods used for broadening the original query depend on the data model. In the last example the system would have to know that "human resources" is a skill more general than "management."

Augmented Answers

The information stored in databases can often be "clustered" into topics, where each topic constitutes a meaningful whole. Quite frequently a query that requests only part of such a topic will be followed by additional queries that will ask for the rest. As an example, consider a query about morning flights from London to Paris. A strict answer would simply list these flights, giving information such as airlines, flight numbers, and departure times. A cooperative answer would also provide *related* information, such as arrival times, prices, and availability. It could also include information on transportation from the airport to the destination city and nearby hotels. It could list early afternoon flights as well, and even suggest comparable alternatives, such as train departures.

Such a technique is described in Ref. *30*, where the data model is a logic formalization of an entity-relationship model with aggregation and generalization relationships. User queries are "completed" in three ways.

1. By relaxing the query specification. This is similar to the techniques described in the subsection "Empty Answers."
2. By "closuring" the requested attributes with additional attributes. The additional attributes originate from entities that participate with the query subject in an aggregation structure (e.g., the hotel attributes).
3. By adding tuples that represent other "objects" altogether. The additional objects originate from entities that participate with the query subject in a generalization structure (e.g., the train tuples). Roughly speaking, this technique expands strict answers by adding both new rows and new columns to the answer tables.

Annotation and Explanation

Like the techniques in the third category, the techniques in the fourth category also provide additional information to users. This additional information is not data, however, but information *about* the data. For example, in response to a query to retrieve the employees who earn more than $50,000, the database system could also inform the user that these employees are the senior managers and engineers, except for Betty (a junior manager who earns more than this amount) and Tom (an engineer who earns less than this amount). Such answers have become to be known as *intensional answers*. Similarly, the database system could annotate the answer with respect to other properties; for example, to qualify portions of the data with respect to their established accuracy.

The section "Detecting Deficiencies in Apparently Good Queries," focused on cooperative techniques that operate on queries that are acceptable to standard database systems, but whereas its three subsections dealt with queries that in some way are deficient (unnecessarily complex, unsatisfiable, or incomplete), the techniques we are about to discuss are intended to provide users with additional insights into the answers that are computed for them by the database system. Our discussion is divided into two topics: intensional answers and meta answers.

INTENSIONAL ANSWERS

An intensional answer is a complement of the conventional answer, comprising either a terse description of the answer or various useful statements that concern the answer.

The term *intensional answer* comes from a distinction often made between the *intension* and the *extension* of a database. The intension of a database is the set of definitions of the data structures for the particular database (the schema). The extension of the database is the set of database values that populate these data structures (the instance). The term intension also refers to various other elements of the definition of the database, such as statements that express relationships that must be satisfied by every extension (constraints) or statements that define new data structures and their extensions in terms of the basic structures (derived relations or inference rules). Specifically, the intension of a relational database includes the definitions of the base relations, the definitions of derived relations, and the integrity constraints. The intension of a logic-based database includes the definitions of the base predicates, the inference rules, and the integrity constraints. In semantically rich or object-oriented models the intension includes the definition of the various classes and their associated hierarchies.

A database query is an intensional statement. Its answer is the extension of this intensional statement. While the intensional information in the database is utilized in the processing of each query, answers are composed entirely of extensional information, consequently the query itself is often the only intensional characterization of the retrieved set of values of which the user is aware. Still, the intensional information in the database may include additional characterizations of the extensional answer. If

this intensional information is derived and retrieved, database answers would gain additional meaning.

Several researchers have recently addressed the issue of intensional answers, (including Refs. 31–37, 27). While all share a common goal, to respond to queries more abstractly by using the intension of the database, the individual approaches are often very dissimilar; they adopt different frameworks (i.e., the data model and its intensional information), they define their intensional answers differently, and they develop their own specific methods for computing them. Space does not permit us to describe individual methods here. A comparative survey of intensional answering may be found in Ref. *38*.

As a broad example, consider a query on the employees who earn more than $50,000. Using different data models and different techniques, possible intensional answers include the following:*

1. The senior managers and engineers, except for Betty (a junior manager who earns more than this amount) and Tom (an engineer who earns less than this amount).
2. The senior managers and their supervisors.
3. The answer given includes all the engineers.
4. All the employees that are included in the answer given are senior.

META ANSWERS

When responding to queries, humans often volunteer additional information *about* their answers. Among other things, they may *qualify* the answer as to its reliability, and they may *characterize* it abstractly. As an example, consider this inquiry about bookstores in Washington (*39*). After listing several bookstores, the persón answering the question might add such comments as the following:

1. "This list is perfect, trust me."
2. "There might be some other bookstores of which I am not aware."
3. "I am confident about all these bookstores, except the last one, which might have been converted into a video boutique."
4. "All these bookstores are located south of M Street."
5. "These bookstores include all those that are located in Georgetown."

The first statement grants assurance that the answer is both sound (all information provided is accurate) and complete (there are no other bookstores in Washington). The second statement states that the answer may be incomplete, while the third statement asserts the soundness of all but the last item. The fourth and the fifth statements provide useful characterizations of the answer. Clearly such characterizations and "quality assurances" are often very valuable to the recipient of the information.**

The various statements about the answer are *properties* of the answer. A relational

*These answers are intended to show the *kinds* of statements that different methods will extract; obviously they are not mutually consistent.

**The last two statements are *intensional statements* similar to statements 3 and 4 in the previous example. Indeed, the scope of meta answers includes intensional answers.

database system that similarly annotates its answers with their properties is Panorama (*39*). The system assumes that various assertions about properties of the data have been stored in the database (meta information). These assertions are then used to infer properties of each answer provided by the system (meta answers). Meta answers are offered to users along with each answer issued, and help them to assess the value and meaning of the information that they receive.

The claimed advantages of Panorama include: (1) it is *extensible* in that it allows users to determine the kinds of properties that the system will maintain and manipulate, (2) it has a built-in mechanism for determining the *relevance* of computed meta information, and (3) it is *efficient*: the number of operations required for meta processing a given query can be expressed as a polynomial in the size of the meta database.

Assuring that Queries Correspond to Objectives

Successful interaction with a database system requires users to have adequate retrieval objectives and the ability to express these objectives in a formal query language. The techniques that were described in this article assist users in meeting these requirements, yet none of these techniques can help a user who has an adequate objective, but implements it *incorrectly* in an adequate query (i.e., the query corresponds to an adequate but *different* objective). Database users would be alerted to such mistakes only if they had a priori knowledge of the form and contents of the answers to their query.

This problem is an instance of the general problem of the so-called logical programming errors, where the programmer writes the wrong program. Detecting such errors is a difficult challenge, and the prevailing approaches are based on testing and proving program correctness. In the case of database queries, however, the problem should be considerably simpler, because database queries operate in a restricted context of specific databases. We offer here two possible techniques for alleviating this problem.

One way to address this problem is to add various *assertions* to the query about *necessary conditions* that must be satisfied by its answer. These assertions are a formal expression of the a priori knowledge mentioned above. Given these assertions, the system can determine automatically when an answer does not meet the assertions and inform the user that either the query or the assertions were misstated. It must be emphasized that because it relies on necessary conditions, this process cannot guarantee that a query that generates an answer that satisfies the assertions is indeed correct. With extensive assertions, however, many query errors can be detected. Examples of assertions include: (1) specification of the range of answer cardinality (e.g., if the answer is empty then the query must be wrong); (2) specification of data that must show up in the answer; and (3) specification of data that must not show up in the answer. As a simple example, a query on the employees that earn more than $50,000 may be accompanied by assertions that "the answer should include at least fifty such employees, and one of them should be Betty." If the answer does not satisfy these assertions then the query or the assertions were misstated.

Another indication that a query might be wrong is that it has never been asked before. An altogether new query—one that is different in substance from all previously asked queries (not merely in its constants or abstraction level)—is suspected of errors, especially if the "history" of previously asked queries has been accumulated over a considerable amount of time.

Prototype Systems

Cooperative techniques have often been isolated strategies for performing specific tasks. These techniques may assume different data models and use different formalisms, consequently the foremost challenge is that of *integration*—to incorporate as many useful techniques as possible into a single elegant paradigm of cooperation.

The proof of any cooperative technique is in its successful implementation and application. Many of the cooperative techniques we surveyed have not progressed beyond modest proof-of-concept demonstrations. A critical challenge is to construct *viable prototypes* that will convince both users and software developers of their usefulness and practicality.

In this section, we describe briefly three prototype systems that enhance database systems with cooperative behavior. In addition to implementing many of the techniques mentioned earlier, each system includes additional features that—while not classified in this article as cooperative techniques per se—are intended to facilitate the use of databases. All three systems are the product of academic research.

FLEX

Flex (*40*) 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 levels of correctness and well-formedness. Each input is "cascaded" through this series of mechanisms until an interpretation is found.

Specifically, four levels can be discerned.

1. At the top level the input is treated as a formal query (the syntax resembles SQL). If found to be correct, the user receives a standard response. Expert users are expected to interact with Flex mostly at this level, which is indistinguishable from standard formal query interfaces.
2. If the query cannot be parsed successfully, an attempt is made to *salvage* it using various techniques that address common formalism errors. If successful, the corrected query is processed and an answer is obtained. Because this second level can deal with queries whose basic structures are recognizable, it assists users with moderate experience or experienced users who make occasional mistakes. Note that at this level there is an increased probability that the derived query would not correspond to the objective of the user.
3. If the query cannot be salvaged, an attempt is made to *synthesize* a common query from

tokens that are recognized in the input. The synthesized query is presented to the user for approval or editing. The assumption here is that while naive users may find it difficult to construct queries from scratch they may be able to comprehend the synthesized queries and possibly even alter them slightly. The probability that queries obtained in this third level would satisfy the objectives precisely decreases even further.

4. Finally, if a query cannot be synthesized (or if the synthesized query is rejected by the user), the system extracts one input token and constructs a frame of information on this token. After examining the information in this frame the user may pick a data element from the frame to construct another frame; in effect, the user is *browsing* in the database.

In addition, Flex also implements the cooperative techniques of empty answers and partial answers, hence 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).

COBASE

CoBase is an ongoing project at the University of California, Los Angeles, under the leadership of Chu (*41*). CoBase uses a data model that extends the relational model with type hierarchies. A unique feature of CoBase is that it is used in conjunction with another system that provides integrated access to a heterogeneous collection of databases (*42*).

Some of the cooperative techniques implemented in CoBase are query broadening (relaxation) for queries that have empty answers, the augmentation of queries with associative (related) information (the determination of the associative information is both context- and user-sensitive), and intensional answers.

CoBase also implements query language extensions that allow users to specify "approximate" values and ask "similar-to" queries. Approximate values (e.g., "find flights departing at *about* 10 A.M.") are translated to intervals of values. Similar-to queries (e.g., "find hospitals similar to St. John's") rank their results based on the similarity of their attributes to the specified value. These features have also been implemented in the database system Vague (*43*).

CARMIN

Carmin is an ongoing project at the University of Maryland under the leadership of Minker to design and implement a cooperative database system (*44*). Two important objectives of Carmin is to integrate different cooperative techniques in a uniform manner and to provide a testbed for new cooperative techniques. Integration is achieved in large part through the use of the logic-based data model that provides a unifying environment for many of the techniques described in this article.

The cooperative techniques embedded in Carmin include query simplification, discovery of misconceptions by checking the satisfiability of the query against the knowledge (constraints), query broadening (relaxation) to identify erroneous presuppositions and to provide partial answers, and construction of explanations for any of

these simplifications or discoveries. Carmin also pays particular attention to issues of computational efficiency involved in producing cooperative responses.

Presently, an experimental prototype of Carmin is available. It is written in Quintus Prolog, and connects to databases stored in the INGRES relational database management system.

Conclusion

In this article we attempted to provide a systematic overview of the various cooperative database techniques that exist. We conclude with a brief discussion of problems that remain to be solved and give a general assessment of the future of this area.

The area of cooperative database systems has attracted much attention, and many clever techniques have been proposed, yet challenges still remain, and we describe here several challenges that we consider to be of primary importance.

We already mentioned in the section "Prototype Systems" that the foremost challenge is that of integration: to bring together various useful techniques in a single paradigm of cooperation. Another important challenge mentioned is the need for viable prototypes that will demonstrate the usefulness and practicality of the proposed techniques. While the prototype systems we described are a step in the right direction, these challenges are yet to be met in full.

Cooperation involves voluntary extension of help. As in other cases of voluntary assistance, it is important to assure that the assistance is *valuable and relevant* to the recipient (or else it will be considered a nuisance and ignored altogether). Many cooperative techniques lack built-in controls for filtering their output to assure that it benefits the specific user with which they communicate. Quite often the need for such controls would not surface in small examples, but only in actual field tests.

The information stored in databases varies with respect to factors such as soundness, completeness, certainty, and currency. Also, the evolution of systems for integrating multiple databases (*45, 46*) raises the inherent problem of inconsistency among answers obtained from multiple sources (*47*). The automatic determination of answer *quality* and the automatic *resolution of inconsistencies* are examples of new areas in which database systems can extend assistance to their users.

The area of cooperative databases has been surveyed in several articles (*48, 49*) and was the focus of several workshops and conferences (*50–54*). These articles and proceedings may provide readers with additional information.

REFERENCES

1. A. K. Joshi, "Mutual Beliefs in Question Answering Systems," in *Mutual Belief*, N. V. Smith, ed. Academic, New York, 1982, pp. 181–197.
2. E. Mays, S. Lanka, A. K. Joshi, and B. L. Webber, "Natural Language Interaction with Dynamic Knowledge Bases: Monitoring as Response," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, Canada, Aug. 24–28, 1981, Morgan Kaufmann, Los Altos, CA, pp. 61–63.

3. B. L. Webber, "Questions, Answers and Responses: Interacting with Knowledge-base Systems," in *On Knowledge Base Management Systems*, Springer-Verlag, Berlin, 1986, pp. 353–402.
4. B. L. Webber and E. Mays, "Varieties of User Misconceptions: Detection and Correction," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* Karlsruhe, Germany, Aug. 8–12, 1983, Morgan Kaufmann, Los Altos, CA, pp. 650–652.
5. S. J. Kaplan, "Cooperative Responses from a Portable Natural Language Query System." *AI*, **19**(2), 165–187 (Oct. 1982).
6. J. D. Ullman, *Database and Knowledge-Base Systems*, vol. I, Computer Science Press, Rockville, MD, 1988.
7. C. Herot, "Spatial Management of Data." *ACM Transac. Database Syst.*, **5**(4), 493–513 (Dec. 1980).
8. R. G. G. Cattell, "An Entity-Based Database Interface," *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Santa Monica, CA, May 14–16, 1980, ACM, New York, pp. 144–150.
   M. Stonebraker and J. Kalash, "TIMBER: A Sophisticated Database Browser," *Proceedings of the Eighth International Conference on Very Large Data Bases*, Mexico City, Sept. 8–10, 1982, Morgan Kaufmann, Los Altos, CA, pp. 1–10.
10. R. G. G. Cattell, "Design and Implementation of a Relationship–Entity–Datum Data Model," technical report CSL-83-4, Xerox Corporation, Palo Alto Research Center, Palo Alto, CA, May 1983.
11. D. Fogg, "Lessons from a 'Living in a Database' Graphical Query Interface," *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Boston, June 18–21, 1984, ACM, New York, pp. 100–106.
12. A. Motro, "BAROQUE: A Browser for Relational Databases." *ACM Transac. Offi. Info. Syst.*, **4**(2), 164–181 (April 1986).
13. T. Learmont and R. G. G. Cattell, "An Object-Oriented Interface to a Relational Database, technical report, Information Management Group, Sun Microsystems, 1987.
14. T. R. Rogers and R. G. G. Cattell, "Object-Oriented Database User Interfaces," technical report, Information Management Group, Sun Microsystems, 1987.
15. A. D'Atri, A. Motro, and L. Tarantino, "ViewFinder: An Object Browser." *Info. Software Tech.*, (forthcoming).
16. X. Pintado and D. Tsichritzis, "An Affinity Browser," in *Active Object Environments*, D. Tsichritzis, ed. Centre Universitaire d'Informatique, Université de Genève, June 1988, pp. 51–60.
17. R. Agrawal, N. H. Gehani, and J. Srinivasan, "OdeView: The Graphical Interface to Ode," *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Atlantic City, NJ, May 23–25, 1990, ACM, New York, pp. 34–43.
18. R. Johnson, M. Goldner, M. Lee, K. McKay, R. Schectman, and J. Woodruff, "USD—A Database Management System for Scientific Research," *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, June 2–5, 1992, ACM, New York.
19. A. D'Atri and L. Tarantino, "From Browsing to Querying." *Data Eng.*, **12**(2), 46–53 (June 1989).
20. A. J̃. Dix and A. Patrick, "Query by Browsing," in *Interfaces to Database Systems*, Proceedings of the Second International Workshop, Lancaster University, July 13–15 1994, P. Sawyer, ed. Workshops in Computing, Springer–Verlag, 1995, pp. 236–248.
21. M. D. Williams, What Makes RABBIT Run? *Internat. J. Man-Machine Stud.*, **21**(4), 333–352 (Oct. 1984).
22. J. M. Janas, "Towards More Informative User Interfaces," *Proceedings of the Fifth International Conference on Very Large Data Bases*, Rio de Janeiro, Oct. 3–5, 1979, ACM, New York, pp. 17–23.
23. U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-Based Approach to Semantic Query Optimization." *ACM Transac. Database Syst.*, **15**(2), 162–207 (June 1990).
24. A. Pirotte, D. Roelants, and E. Zimanyi, "Controlled Generation of Intensional Answers." *IEEE Transac. Knowl. Data Eng.*, **3**(2), 221–236 (June 1991).
25. F. Corella, S. J. Kaplan, G. Wiederhold, and L. Yesil, "Cooperative Responses to Boolean Queries," *Proceedings of the IEEE Computer Society First International Conference on Data Engineering*, Los Angeles, April 24–27, 1984, IEEE Computer Society, Washington, DC, pp. 77–85.
26. A. Motro, "SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems." *ACM Transac. Offi. Info. Syst.*, **4**(4), 312–330 (Oct. 1986).

27. W. W. Chu, R.-C. Lee and Q. Chen, "Using Type Inference and Induced Rules to Provide Intensional Answers," *Proceedings of the IEEE Computer Society Seventh International Conference on Data Engineering*, Kobe, Japan, April 8–12, 1991, IEEE Computer Society, Washington, DC, pp. 396–403.
28. T. Gaasterland, P. Godfrey, and J. Minker, "Relaxation as a Platform for Cooperative Answering." *J. Intell. Info. Syst.*, **1**(3/4), 293–321 (Dec. 1992).
29. A. Gal, *Cooperative Responses in Deductive Databases*, Ph.D. thesis, Dept. of Computer Science, University of Maryland, 1988.
30. F. Cuppens and R. Demolombe, "Cooperative Answering: A Methodology to Provide Intelligent Access to Databases," *Proceedings of the Second International Conference on Expert Database Systems*, Tysons Corner, VA, April 25–27, 1988, George Mason University, Fairfax, VA, pp. 333–353.
31. L. Cholvy and R. Demolombe, "Querying a Rule Base," *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April 1–4, 1986, Institute of Information Management, Technology and Policy, University of South Carolina, Columbia, pp. 365–371.
32. T. Imielinski, "Intelligent Query Answering in Rule Based Systems." *J. Logic Prog.*, **4**(3), 229–257 (Sept. 1987).
33. C. D. Shum and R. Muntz, "Implicit Representation for Extensional Answers," *Proceedings of the Second International Conference on Expert Database Systems*, Tysons Corner, VA, April 25–27, 1988, George Mason University, Fairfax, VA, pp. 257–273.
34. C. D. Shum and R. Muntz, "An Information-Theoretic Study on Aggregate Responses," *Proceedings of the Fourteenth International Conference on Very Large Data Bases,* Los Angeles, Aug. 25–28, 1988, Morgan Kaufmann, Los Altos, CA, pp. 479–490.
35. A. Pirotte and D. Roelants, "Constraints for Improving the Generation of Intensional Answers in a Deductive Database," *Proceedings of the IEEE Computer Society Fifth International Conference on Data Engineering*, Los Angeles, Feb. 6–10, 1989, IEEE Computer Society, Washington, DC, pp. 652–659.
36. A. Motro, "Using Integrity Constraints to Provide Intensional Responses to Relational Queries," *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, Aug. 22–25, 1989, pp. 237–246.
37. T. Andreasen, "Semantic Query Answering," *Proceedings of COMAD 90*, New Delhi, India, N. Prakash, ed. McGraw Hill, New Delhi, 1990.
38. A. Motro, "Intensional Answers to Database Queries." *IEEE Transac. Knowl. Data Eng.*, **6**(3), 444–454 (June 1994).
39. A. Motro, "Panorama: A Database System that Annotates Its Answers to Queries with Their Properties." *J. Intell. Info. Syst.*, **7**(1), (1996).
40. A. Motro, "FLEX: A Tolerant and Cooperative User Interface to Databases." *IEEE Transac. Knowl. Data Eng.*, **2**(2), 231–246 (June 1990).
41. W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson, "Cobase: A Scalable and Extensible Cooperative Information System." *J. Intell. Info. Syst.*, **6**(2/3), 223–259 (June 1996).
42. Y. Arens, C. A. Knoblock, and W.-M. Shen, "Query Reformulation for dynamic Information Integration." *J. Intell. Info. Syst.*, **6**(2/3), 99–130 (June 1996).
43. A. Motro, "VAGUE: A User Interface to Relational Databases that permits Vague Queries." *ACM Transac. Offi. Info. Syst.*, **6**(3), 187–214 (July 1988).
44. P. Godfrey, J. Minker, and L. Novik, "An Architecture for a Cooperative Database System," *Applications of Databases: Proceedings of the First International Conference*, Vadstena, Sweden, June 21–23, 1994, Lecture Notes in Computer Science, vol. 819, Springer-Verlag, pp. 3–24.
45. A. R. Hurson, M. W. Bright, and S. H. Pakzad, eds., *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, IEEE Computer Society Press, Los Alamitos, CA, 1994.
46. G. Wiederhold, ed., "Intelligent Integration of Information." *J. Intell. Info. Syst.*, **6**(2/3) (June 1996).
47. A. Motro, "A Formal Framework for Integrating Inconsistent Answers from Multiple Information Sources," technical report ISSE-TR-93-106, Dept. of Information and Software Systems Engineering, George Mason University, Oct. 1993.
48. T. Gaasterland, P. Godfrey, and J. Minker, "An Overview of Cooperative Answering." *J. Intell. Info. Syst.*, **1**(2), 123–158 (Oct. 1992).
49. A. Motro, "Cooperative Database Systems," *Internat. J. Intell. Syst.*, **11**(10), 717–732 (Oct. 1996).

50. R. Demolombe and T. Imielinski, eds., *Nonstandard Queries and Nonstandard Answers*, Studies of Logic and Computation, vol. 3, Oxford Science Publications, Clarendon, Oxford, 1994.
51. *Proceedings of FQAS-94, Workshop on Flexible Query Answering Systems* Roskilde, Denmark, Nov. 14–16, 1994, Datalogiske skrifter 58, Roskilde University.
52. *Proceedings of FQAS-96, Workshop on Flexible Query Answering Systems*, Roskilde, Denmark, May 22–24, 1996, Datalogiske skrifter 62, Roskilde University.
53. *Flexible Query Answering Systems*, Kluwer, 1997.
54. *Flexible Query Answering Systems*, Proceedings of the Third International Conference, Roskilde, Denmark, May 13–15, 1998, Lecture Notes in Artificial Intelligence no. 1495, Springer-Verlag.

AMIHAI MOTRO