SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems

AMIHAI MOTRO
University of Southern California

Every information system incorporates a database component, and a frequent activity of users of information systems is to present it with queries. These queries reflect the presuppositions of their authors about the system and the information it contains. With most query processors, queries that are based on erroneous presuppositions often result in null answers. These fake nulls are misleading, since they do not point out the user's erroneous presuppositions (and can even be interpreted as their affirmation). This article describes the SEAVE mechanism for extracting presuppositions from queries and verifying their correctness. The verification is done against three repositories of information: the actual data, their integrity constraints, and their completeness assertions. Consequently, queries that reflect erroneous presuppositions are answered with informative messages instead of null answers, and user–system communication is thus improved (an aspect that is particularly important in systems that often are accessed by naive users). First, the principles of SEAVE are described abstractly. Then, specific algorithms for implementing it with relational databases are presented, including a new method for storing knowledge and an efficient algorithm for processing queries against the knowledge.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models*

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Cooperative user interface, database, database completeness, database integrity, erroneous presupposition, query failure, query generalization, relational database

1. INTRODUCTION

A basic requirement of any user–system interface is that it be able to reject (with proper explanation) all improper input. Interfaces differ, however, in their *definition* of improper input. Some interfaces check only basic rules of *syntax*; others may examine, to varying degrees, the *semantics* of the input. Clearly, any system action that is based on input that should have been rejected is bound to be meaningless and even misleading. Thus, at the risk of oversimplification, one could state that an interface that rejects more inputs as improper is a "better" interface.

User interfaces to databases have only limited rejection capabilities. Often, the only rejections they are capable of (in addition to those that are based on syntax)

This work was supported in part by an Amoco Foundation Engineering Faculty Grant.
Author's address: Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

are based on the *schema.* Most database systems employ data models that distinguish between a generic description of the data, called schema, and the actual data that populate the schema. In such systems, requests for data usually must be stated in terms of the schema. For example, to retrieve the names of the employees who earn $40,000 a year from a relational database, one submits a query such as "`retrieve from relation` EMPLOYEE `all values of attribute` E-NAME `where attribute` SALARY `has value 40,000`." In this query, EMPLOYEE, E-NAME, and SALARY are schematic constructs. If there is no relation EMPLOYEE in the database, or relation EMPLOYEE does not include both attributes E-NAME and SALARY, the system will reject this query on grounds of schema errors.

We prefer to think of a user who submitted a query that contained schema errors as having had *erroneous presuppositions* regarding the structure of the database. The user perceived the database as having a particular structure and formulated a query accordingly; the query then reflected this presupposition. When processing the query, the system attempted to *verify* this presupposition against the actual schema, failed, and notified the user. Thus, schema-based rejections can be regarded as detections of erroneous presuppositions.

Assume now that relation EMPLOYEE also has an attribute DEPARTMENT, and consider this query to list the names of the employees in the Shipping Department who earn $40,000: "`retrieve from relation` EMPLOYEE `all values of attribute` E-NAME `where attribute` DEPARTMENT `has value Shipping and attribute` SALARY `has value 40,000`." Assume that there is no Shipping Department (i.e., `Shipping` is not a value of DEPARTMENT). Most query processors will simply return a null answer, which may then be interpreted to mean that no employees in the Shipping Department earn $40,000. However, it is obvious that this query reflects a presupposition that a Shipping Department does exist, and a more meaningful reaction would be to reject the query, pointing out that this presupposition is erroneous.

These two examples demonstrate our approach. Each query submitted to a database reflects presuppositions of the user who composed it ("There is a relation EMPLOYEE with attributes E-NAME and SALARY," "There is a department called Shipping"). By extracting such presuppositions from the query and attempting to verify them against the system's own knowledge, the system can improve its rejection capabilities. This article describes such a mechanism.

The process of extracting presuppositions and attempting to verify them involves additional computation. Clearly, if the query is attempted first and matches some data, we can safely assume that the presuppositions of the user who submitted it are correct. Therefore, it is only when the query *fails* to match any data that we attempt to verify the presuppositions behind it. Thus, *null answers* trigger the mechanism.[1]

There are times when null answers are *genuine.* Such is the case, for example, when the previous database is queried about the employees of the Personnel Department who earn less than $12,000. If every employee in the Personnel Department earns more than $12,000, the null answer is appropriate.

[1] Actually, the evaluation of a query against the database is considered the first test in the query verification process. If the query fails this test (returns a null answer), additional tests are performed.

Unfortunately, the same null answer will be given when that query is repeated for the employees of the Shipping Department (which does not exist). Indeed, when asked these two questions, a knowledgeable person would probably respond differently to each one: The response to the first question would be "There are no employees in the Personnel Department who earn less than $12,000," or simply "There aren't any"; the response to the second question would be "There is no Shipping Department," or simply "You don't know what you are talking about." The latter (somewhat rude) response is quite correct, since the person who posed the query implied that such a department does exist. In other words, some null answers can be attributed to an erroneous presupposition on behalf of the user. These null answers are referred to as *fake*.

Clearly, a fake null is unsatisfactory, since it may be interpreted by the user as a genuine null (and consequently also as an affirmation of the user's presuppositions). At other times, when the user knows for certain that the query should have matched some data, the user interprets the fake null as an error message, indicating that the query did not express the user's intentions correctly. Such error messages are too general and not at all informative.

Even genuine null answers are often disturbing, since the information they provide amounts to a "shrug." This response contrasts with human behavior in which a negative answer is usually accompanied by some additional information. For example, when presented with the first of the above questions, a person might reply: "But there are some who make less than $15,000." In general, such answers are helpful, since they inform the person asking that the question was indeed meaningful, and that its failure was genuine, not the result of some erroneous presupposition. More important, such answers tend to delimit the scope of the failure; in the previous example a negative answer could have been caused by a more fundamental inability to satisfy the question, and the person asking could be left wondering about the real cause of the negative answer. Finally, sometimes such answers anticipate subsequent questions, since often negative answers trigger follow-up questions.

Efforts to address the problem of null answers that are results of erroneous presuppositions can be traced back to the system CO-OP, designed by Kaplan [11], which implemented some of the conventions of cooperation in human discourse. These include *corrective* responses that detect erroneous presuppositions and *suggestive* responses that anticipate follow-up queries. CO-OP was designed for natural language interaction and relied on domain-specific knowledge. More recently, Corella et al. [4] and Motro [15] discussed similar techniques in the environment of a typical database management system, which has only formal language interfaces and no domain-specific knowledge.

In all these efforts the basic approach is to follow up a query that failed with several more general queries. If even these fail, then the conclusion is that some of the presuppositions of the user who composed the original query are erroneous. The justification for this approach is in a heuristic called *minimal uncertainty*, which we define in [15]:

—While users expect that their queries may possibly have null answers, they tend to be confident that every more general query would not have failed.

Under this heuristic, the more general queries become indicators of the presuppositions of the user. Thus, a query that fails produces a genuine null if and only if every more general query succeeds. Otherwise, the null is fake, and erroneous presuppositions are reported back.

However, it is entirely possible that the user's presuppositions are all correct, and the database simply does not include any data to satisfy the query and some of its generalizations. In other words, on more general queries the mechanism interprets "there are no data" as "there could be no data." By using additional information available in databases, such as integrity constraints and completeness information, these methods can be improved so that more erroneous presuppositions can be pointed out with more authority.

This article is divided into two parts. The first part describes the principles of a new mechanism called SEAVE (Supposition Extraction And VErification) for detecting erroneous presuppositions. These principles are discussed abstractly and are independent of any particular database model. The second part focuses on the relational database model. It presents specific algorithms for implementing SEAVE in relational systems, including a new method for storing database knowledge and an efficient algorithm for processing queries against the knowledge.

2. THE PRINCIPLES OF SEAVE

The process of extracting presuppositions from queries and verifying them against the database knowledge involves three principal issues: how to determine the presuppositions of the user, what knowledge does a database system have, and how to verify effectively the presuppositions against the knowledge. These issues are discussed below.

2.1 User Presuppositions

The standard interaction between users and databases is an iterative process in which users present queries and the database management system returns answers (and/or messages). Since queries are the only input users contribute, they are the only source from which the system may infer their presuppositions.

Queries are requests to retrieve from the database a particular set of facts. Obviously, a user who submits a query believes that possibly this set is nonempty (otherwise, why ask). This leads to the following simple method for inferring presuppositions from queries:

—Each query reflects a presupposition that the condition it expresses is *plausible* (will possibly succeed).

Thus, given the query "`retrieve all facts that satisfy` *condition*," we infer the presupposition "`there may be facts that satisfy` *condition*," or simply, "*condition* `is plausible`." Because of this tight correspondence between queries and presuppositions, these terms will sometimes be used interchangeably. The presuppositions we extract are based on a single user input. In this connection we note that there are systems for understanding natural language that gather information from earlier inputs (e.g., [9]); some even maintain

a model of the user's world (e.g., [1], [13], and [22]). However, these kinds of models are outside the scope of this work.

Consider now a query about all the boys who have an older sister. It reflects a presupposition of the user that there may be boys who have an older sister. However, it also reflects some additional presuppositions, such as: there may be boys who have a sister; there may be boys who have a female relative; there may be children who have relatives; and so on. Compared with the original presupposition (which was inferred from the query), these presuppositions are *more general* (i.e., *weaker*) and can be inferred from the original presupposition by an appropriate procedure. Hence:

—From each presupposition more general presuppositions may be inferred.

A presupposition may be generalized in different *directions* and to varying *levels*. Recall the previous presupposition that there are boys who have an older sister. One direction is to substitute for "boys" a more general concept, such as "children"; this direction can be pursued further, and "children" may be replaced by "persons." Another direction is to substitute "older female relative," or even "older relative," for "older sister." In effect, the generalizations of a given presupposition form a Boolean lattice.

We have emphasized that a user who submits a query *believes* that it is plausible. But how *strong* is this belief? In other words, how confident is the user about such presuppositions? Although it is impossible to answer this question directly, it is safe to assume that users are more confident about more general presuppositions:

—Given two presuppositions (inferred from a query), the user is more confident about the more general presupposition.

2.2 Database Knowledge

To test presuppositions, the database management system applies its knowledge about the concepts mentioned in the presupposition. This knowledge may be available in various forms. We consider here three different kinds of knowledge available in databases: *facts*, *integrity constraints*, and *completeness assertions*.

Facts constitute the bulk of the database, and the main purpose of a database system is to provide access to these facts. Facts are a low-level description of the environment, since the information encapsulated in each fact applies only to the individual real-world objects named in the fact.

Integrity constraints [6] describe relationships that must be maintained among the facts. Usually, these constraints are used to monitor the consistency of the database and reject all updates that would violate them. In general, integrity constraints are *template* statements (i.e., they make use of variables), and the information encapsulated in each constraint usually applies to a multitude of facts.

Completeness assertions, introduced in [16], describe the subsets of the database that include a representation of every real-world occurrence (subsets that are "closed world"). With completeness information a database system can determine whether each answer to a user query is complete (i.e., whether all the real-world occurrences are represented), or whether any *subsets* of it are complete

(i.e., whether all the real-world occurrences that satisfy some additional constraints are represented). Often, when the user receives the answer to a query, there is uncertainty as to the *quality* of the answer (is it accurate? is it complete?). Answers that are accompanied by statements about their completeness are, therefore, more meaningful.

Finally, we note that in addition to facts, integrity constraints, and completeness assertions, database knowledge may be represented in additional forms, such as inference rules [7] or exceptional information [2].

2.3 Testing Presuppositions

In correspondence with the three kinds of database knowledge described above, we define three tests for determining whether a presupposition extracted from a query is correct:

(1) *Test against facts.* This is the simplest test. Recall that each presupposition is actually a belief that a query may possibly succeed. If the query is evaluated and fails, then the presupposition is rejected on the grounds that *the database knows of no facts to support it.*

(2) *Test against completeness assertions.* The query is checked for completeness of its answer. If it fails but can be shown to have a complete answer, then the presupposition is rejected on the grounds that *there are no facts to support it.*

(3) *Test against integrity constraints.* The query is checked for validity. If it violates any integrity constraint, then the presupposition is rejected on the grounds that *there may be no facts to support it.*

These individual tests can be performed independently. However, note that, if a presupposition can be verified against the facts, any further testing is pointless. A procedure for testing individual presuppositions that combines all three tests is outlined below. This procedure is referred to as the *Supposition Tester* (*ST*).

Consider a presupposition p submitted for testing. First, p is tested against the facts. If it matches some facts in the database, the process terminates with the verdict "p verified." If p does not match any facts, it is tested for completeness. If p is shown to be complete, the process terminates with the verdict "p rejected because there are no facts to support it." If p cannot be shown to be complete, it is tested for validity. If p is shown to violate some integrity constraint, then the process terminates with the verdict "p rejected because there may be no facts to support it." If p cannot be shown to be invalid, then the process terminates with the verdict "p rejected because the system knows of no facts to support it." A schematic diagram of ST is shown in Figure 1.

2.4 The Verification Mechanism

As already mentioned, each query reflects a presupposition, and this presupposition implies several, more general (weaker) presuppositions. When a presupposition is rejected by ST, then every more specific (stronger) presupposition will also be rejected, and when a presupposition is verified by ST, then every more general presupposition will also be verified.

Clearly, if two presuppositions are rejected, and one is more general than the other, then the rejection of the more specific presupposition is insignificant.

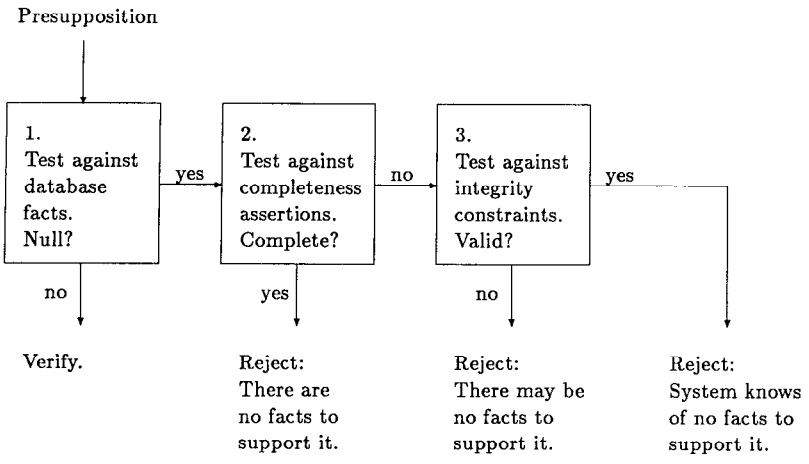

Fig. 1. A schematic diagram of ST.

Hence this definition:

—An erroneous presupposition is *significant* if and only if every more general presupposition can be verified.

Given a user query, the goal is then to report to the user all the erroneous presuppositions that are significant, that is, to detect the rejected generalizations of the original presupposition for which every more general presupposition can be verified. Toward this goal, we define the concept of a *minimal generalization*:

—Given a presupposition p and a presupposition p' that is more general (weaker), p' is *minimally more general* than p, if every other generalization of p is also more general than p'.

A *Supposition Generalizor* (*SG*) is a component that, given an input presupposition, generates a set of output presuppositions, all minimally more general that the input presupposition. To perform this task, SG incorporates various strategies, depending on the data model used by this system. For example, in a relational database system, generalization may be accomplished by weakening mathematical conditions or by deleting conjuncts from queries. In data models that incorporate a type hierarchy (e.g., [8], [18]), type substitution may be performed.

As an example, consider a database with information on employees, including their name, age, and salary, and a query to list all the female employees under 30 years of age who earn at least $40,000 a year. Figure 2 shows a portion of the lattice of presuppositions, generated by a particular supposition generalizor. Nodes indicate presuppositions, and arcs indicate generalization relationships. A presupposition that there may be employees whose age is under x, whose sex is y, and whose yearly salary is at least z, is denoted (x, y, z). The symbol * indicates any value; once it appears in a presupposition it cannot be generalized any further.

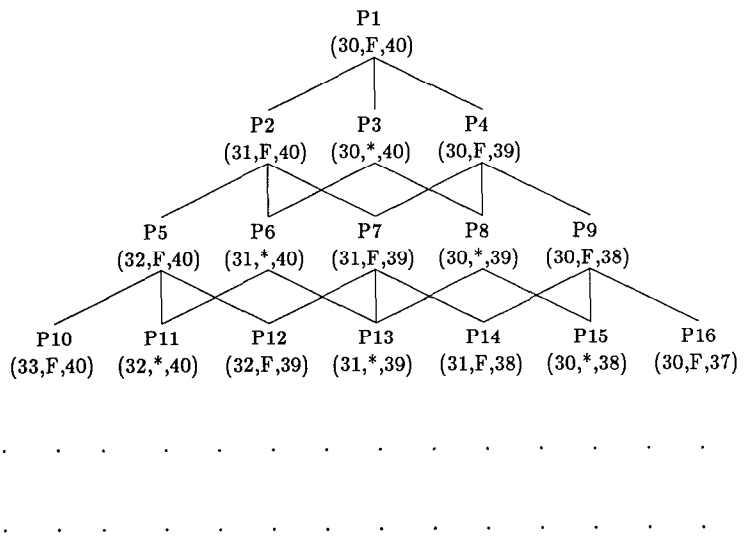

Fig. 2. Portion of a presupposition lattice.

```
SEAVE(p);
var p, q: presupposition;
var P: set of presupposition;
var significant: boolean;
local q, P, significant;
begin
  if ST(p)
    then
      return true;
    else
      begin
        significant:=true;
        P:=SG(p);
        for each q in P do
          significant:=significant and SEAVE(q);
        if significant
          then
            print("Erroneous and significant presupposition:", q);
        return false;
      end;
end.
```

Fig. 3. The SEAVE algorithm.

The SEAVE mechanism combines the Supposition Generalizor and the Supposition Tester. Given a query, it derives the set of all erroneous presuppositions that are significant. Figure 3 describes the algorithm in an ALGOL-like language. The algorithm calls two other procedures: ST receives a presupposition and returns *true* if the presupposition can be verified and *false* otherwise; SG receives a presupposition and returns a set of minimally more general presuppositions.

The output of the algorithm is a list of all the erroneous presuppositions that are significant.

In the example of Figure 2, assume that presupposition P1 is rejected. Among its generalizations, assume that P3 is verified (and, therefore, also P6, P8, P11, P13, and P15), but P2 and P4 are rejected. P5 and P7 are then verified (and, therefore, also P10, P12, and P14), but P9 is rejected. Finally, P16 is verified. The two erroneous and significant presuppositions are P2 and P9; they state:

(1) There may be female employees under 31 who earn at least $40,000.
(2) There may be female employees under 30 who earn at least $38,000.

A presupposition is verified after the corresponding query matches some facts. Consequently, the process of detecting the erroneous and significant presuppositions also provides *partial answers* to the original query. In the above example, P3, P5, P7, and P16 were verified. In the process, these partial answers were computed:

(1) all employees under 30 who earn at least $40,000,
(2) all female employees under 32 who earn at least $40,000,
(3) all female employees under 31 who earn at least $39,000,
(4) all female employees under 30 who earn at least $37,000.

These partial answers can be delivered to the user as "the best the system could do" to satisfy the query about all female employees under 30 who earn at least $40,000.

3. ALGORITHMS FOR A RELATIONAL IMPLEMENTATION

The principles of SEAVE can be adapted for any database model. To demonstrate SEAVE in more detail, we present here specific algorithms for implementing it with relational databases. The relational model was selected primarily because of its simplicity and widespread use. A particular concern, which influenced the solutions, was to keep the cost of verification low.

We assume queries are submitted in QUEL's retrieve statement [20]. As an example, the following query retrieves from the database described in Figure 4 the names and salaries of all employees who are managed by Lucy and know editing:

```
range of e is EMPLOYEE
range of d is DEPARTMENT
range of s is SKILL
retrieve (e.E-NAME, e.SALARY)
where e.E-NAME = s.E-NAME
  and s.S-NAME = "Editing"
  and e.DEPARTMENT = d.D-NAME
  and d.MANAGER = "Lucy".
```

We assume that the **where** clause includes only conjunctions and disjunctions of atomic formulas. Each atomic formula is a comparison between a variable and a constant or between two variables. The comparator is either $=$ or $\neq$. For variables that range over numeric attributes it can also be one of these: $>, \geq, <, \leq$.

```
EMPLOYEE = E-NAME, SALARY, DEPARTMENT
DEPARTMENT = D-NAME, MANAGER, BUDGET
SKILL = E-NAME, S-NAME, LEVEL
```

Fig. 4. Schema of database STUDIO.

3.1 The Supposition Generalizor

Generalization of presuppositions is performed by weakening mathematical conditions or removing nonmathematical conditions altogether. It is assumed that for each numeric attribute, three values are available: a minimum value and a maximum value to indicate the allowable range of this attribute, and a step value to specify the minimal weakening of a mathematical condition. For example, the attribute SALARY may have minimum 10,000, maximum 100,000, and step 1000.[2] Given a query, the query generalizor identifies the subformulas of the **where** clause of the type $A\ \theta\ c$, where A is a variable and c is a constant. Each such subformula is a basis for a generalization, according to the following process:

Assume that A is a numeric attribute with minimum m_1, maximum m_2, and step d and that c is a number. If θ is $>$, this subformula will be replaced by $A > c'$, where $c' = \max\{c - d, m_1\}$. If θ is $<$, it will be replaced by $A < c'$, where $c' = \min\{c + d, m_2\}$. Analogous replacements will be performed when θ is $\geq$ or $\leq$. If θ is $=$, then it will be replaced by a conjunction of two subformulas: $A \geq c'$ *and* $A \leq c''$, where $c' = \max\{c - d, m_1\}$ and $c'' = \min\{c + d, m_2\}$. If θ $\neq$, then it will simply be deleted.

If A is a nonnumeric attribute and c is a string, then the subformula is deleted. Note that if the database incorporates *data metrics*, as suggested in [17], then queries with nonnumeric attributes may be generalized in finer steps. The subformula $A \in C$, where C is a set of values *close* to c, would be substituted for $A = c$.

3.2 A Unified Representation of Database Knowledge

Of the three kinds of knowledge discussed in Section 2.2, the representation of database facts requires no special attention, since they are simply the tuples of the relations. The representation of completeness assertions and integrity constraints is discussed below. This representation is an extension of the representation of completeness assertions that we described in [16].

Since the intention is to check user queries against completeness assertions and integrity constraints, an obvious candidate for representing queries, assertions, and constraints is first-order predicate logic. However, testing queries against the knowledge would then require general theorem proving. This solution is not very desirable because of its complexity and undecidability. We concur with [14] on the trade-off between the expressivity of the language for representing knowledge and the complexity of the reasoning procedures, and choose to adopt a more restricted language. This language would be simpler to manipulate, yet powerful enough to be useful.

Behind the language is the observation that both completeness assertions and integrity constraints are actually specifications of *database subsets* (i.e., derived

[2] The allowable range is similar to domain integrity constraints [5].

relations). A completeness assertion is a definition of a subset that is known to include a representation of every real-world occurrence. Such subsets are called *complete* subsets. Integrity constraints are often of the type "if $condition_1$, then $condition_2$," where both conditions describe relationships among data items. Each such statement can be restated as "the subset of facts, for which $condition_1$ holds and $condition_2$ does not hold, is empty." Such subsets are called *null* subsets. Consequently, the formalisms necessary to express completeness assertions and integrity constraints can be provided by any query language.

In this article we consider only completeness assertions and integrity constraints (collectively referred to as *knowledge statements*) that specify database subsets of the following kind:[3]

$$\{a_1, \ldots, a_n \mid (\exists b_1) \cdots (\exists b_m) Q_1 \wedge \cdots \wedge Q_k\}$$

where the Q's are of either of these two forms:

(1) $R(c_1, \ldots, c_r)$, meaning that the tuple $(c_1, \ldots, c_r)$ is asserted to be in relation R; the c's are either among the a's or b's or are constants.

(2) $c \theta d$, where c is among the a's or b's, d is among the a's or b's or is a constant, and θ is one of $=, \neq, >, \geq, <, \leq$.

These statements can be applied to describe either completeness assertions or integrity constraints. For example, in the previous database, the completeness of the set of all names of employees in Construction who are electricians is stated as follows:

$$\{a \mid (\exists b_1)(\exists b_2)\text{EMPLOYEE}(a, b_1, \text{"Construction"}) \wedge \text{SKILL}(a, \text{"Electrical"}, b_2)\}.$$

And a constraint that all employees who know editing earn at least $30,000:

$$\{a \mid (\exists b_1)(\exists b_2)(\exists b_3)\text{EMPLOYEE}(a, b_1, b_2) \wedge \text{SKILL}(a, \text{"Editing"}, b_3) \wedge (b_1 < \text{"30,000"})\}.$$

Note that, while the completeness assertion directly describes the set of tuples that is complete, the integrity constraint must be restated to describe a set of tuples that is empty.

For knowledge statements from this family we have developed a representation that resembles regular data tuples. As we point out later in this section, this approach provides important advantages. This method recalls the representation of QBE queries in skeleton tables [23].

A knowledge statement from this family is represented as several *knowledge tuples*. For each subformula $R(c_1, \ldots, c_r)$, a tuple, obtained from $(c_1, \ldots, c_r)$, is stored in R. The c's that are variables (a's or b's) that appear only once in this statement are replaced with blanks; the c's that are a's are suffixed with *. For each subformula $c\ \theta\ d$, a tuple (c, θ, d) is stored in a special new relation CONDITION. When θ is $=$, the tuple in CONDITION may be discarded and the variable c replaced with d throughout the other knowledge tuples. Consequently, each component of a knowledge tuple may be either a constant or a variable or a

[3] This family of subsets is often used for querying, where it is known as the family of *conjunctive queries* [21].

EMPLOYEE

	E-NAME	SALARY	DEPARTMENT
	Betty	38,000	Production
	John	26,000	Construction
	Tom	32,000	Marketing
	Mary	34,000	Production
C1	*		
C2	x^*		Construction
I1	y^*	z	
I2	u^*		v

SKILL

	E-NAME	S-NAME	LEVEL
	Betty	Cinematography	2
	John	Electrical	3
	John	Directing	1
	Tom	Operating	2
	Mary	Acting	2
	Mary	Editing	3
C2	x	Electrical	
I1	y	Editing	
I2	u	Editing	

DEPARTMENT

D-NAME	MANAGER	BUDGET
Construction	John	3,000,000
Marketing	Tom	2,750,000
Production	Betty	4,250,000

CONDITION

	A	Θ	B
I1	z	$<$	30;000
I2	v	$\neq$	Production

Fig. 5. A database extended with knowledge statements.

blank, and each may possibly be suffixed by *. A constant imposes a restriction on the values that an attribute may have; a variable forces this value to be identical to the value of another attribute; a blank is an attribute whose value is irrelevant to this statement. A * indicates an attribute on which knowledge is stated.

We assume that a variable name is not used in more than one knowledge statement, and that tuples of completeness assertions are distinguishable from tuples of integrity constraints. A relation with knowledge tuples is called an *extended* relation.

As an example, Figure 5 shows four knowledge statements, together with a small instance of the database. They state that the set of names of employees is complete (C1), that the set of names of employees in Construction who are electricians is complete (C2), that every employee who knows editing earns at least $30,000 (I1), and that every employee who knows editing must be in Production (I2).

Note that when an attribute on which knowledge is asserted (i.e., an attribute that has * in the knowledge tuple) functionally determines another attribute, then the same knowledge may be asserted on the other attribute as well. For example, since DEPARTMENT is functionally dependent on E-NAME, if the set of all employee names is complete, so is the set of all departments for which employees work.

This method for storing knowledge has several advantages. First, the specification of knowledge statements using QBE-like notation is very intuitive. Second, storing the knowledge does not require any new data structures. Third, the knowledge may be updated with the same tools used to update the data. But the greatest advantage is in the testing of presuppositions. With this storage strategy, it is possible to develop a method for performing all three tests (Figure 1) concurrently, so that, in effect, the second and third tests become by-products of the standard query evaluation (the first test). Such a method is described next.

3.3 A Unified Test of Presuppositions

As mentioned above, knowledge statements define database subsets that are either complete or null. A given query also defines a database subset. If an answer to a query is contained in a complete subset, then the answer provided is complete (and if it is null, then the presupposition fails the second test). If an answer to a query is contained in a null subset, then the query is a violation (and the presupposition fails the third test). Therefore, the second and third tests can be performed by a procedure to detect whether the answer to a given query is contained in one of the subsets defined by the knowledge statements.

Our approach is to extend the operations that manipulate relations during query processing so that they preserve the knowledge in extended relations. Each relational operator would manipulate the knowledge tuples so that the completeness tuples in the resulting relation will, indeed, describe subsets of the result that are complete, and the integrity tuples in the resulting relation will, indeed, describe subsets of the result that are null. Consequently, when the final relation (the answer to the query) is derived, its knowledge tuples will describe subsets that are complete or null. This relation need only be checked for knowledge tuples that have * in every component. A completeness tuple of this kind states that the answer is contained in a complete subset (the data tuples in the result, if any, form a complete subset). An integrity tuple of this kind states that the answer is contained in a null subset (if integrity is enforced, then the result should not contain any data tuples).

The QUEL queries we assumed can be implemented with three relational operations: Cartesian product, selection, and projection. The extension of these operators to preserve knowledge is discussed below. (Note that relation CONDITION is not manipulated directly; it only assists in performing selections.)

Note that the meaning of each knowledge statement depends on the semantics of the relations that are involved. Therefore, the semantics of all database relations must be determined before knowledge statements are defined. To show that knowledge is preserved after each extended operation, it is important to determine the semantics of the output relations in terms of the semantics of the input relations. For example, when a selection is applied to a relation that models a particular real-world concept, the result no longer models the original concept but a more restricted one. Thus, if EMPLOYEE models all employees, then $\sigma_{\text{DEPARTMENT}=\text{Construction}}(\text{EMPLOYEE})$ models only the employees in Construction. Consequently, although both relations may have the same attributes, a knowledge tuple in the output relation asserts something different from the same knowledge tuple in the input relation.

3.3.1 *Cartesian Product.* Let R and S be two extended relations, and let r and s denote, respectively, completeness (integrity) tuples from R and S. Let rs denote their concatenation, and let – denote a tuple of blanks. The Cartesian product of R and S includes the usual data tuples obtained from the data tuples of R and S, plus the following knowledge tuples:

(1) If both r and s are completeness (integrity) tuples, then include the completeness (integrity) tuple rs.

(2) If r is a completeness (integrity) tuple, then include the completeness (integrity) tuple r–.

(3) If s is a completeness (integrity) tuple, then include the completeness (integrity) tuple –s.

In the example of Figure 5, consider the Cartesian product of EMPLOYEE and SKILL. The three completeness tuples in these relations will be carried over to the product by extending each with blanks, and each of the two completeness tuples of EMPLOYEE will be concatenated with the completeness tuple of SKILL, yielding a total of five completeness tuples. Similarly, the product will include eight integrity tuples.

3.3.2 *Selection.* We assume that the selection formula is a single comparator (the generalization to more complex formulas is straightforward). Let R be an extended relation, and consider a selection condition $A\ \theta\ a$ that compares an attribute of R with a constant. This selection from R by this condition includes the usual data tuples, plus the knowledge tuples for which either of the following is true:

(1) The value in attribute A is a constant a', and $a'\ \theta\ a$.
(2) The value in attribute A is a variable x that is linked only to an entry $x\ \phi\ a'$ in relation CONDITION, and $x\ \phi\ a'$ implies $x\ \theta\ a$.
(3) The value in attribute A is a variable or a blank suffixed by *.

In the example, consider the selection DEPARTMENT = "Construction" from relation EMPLOYEE. The knowledge tuples C2 and I2 fall, respectively, under the first and second cases and are, therefore, selected.

Similarly, consider a selection condition $A\ \theta\ B$ that compares two attributes of R. Some of the cases in which a knowledge tuple is selected are

(1) The values in attributes A and B are constants a and b, respectively, and $a\ \theta\ b$.
(2) The value in attribute A is a variable x that is linked only to an entry $x\ \phi\ a'$ in relation CONDITION, the value in attribute B is a constant b, and $x\ \phi\ a'$ implies $x\ \theta\ b$.
(3) The values in attributes A and B are variables or blanks suffixed by *.

3.3.3 *Projection.* We consider only projections that remove a single attribute (the treatment of general projections is similar). Let R be an extended relation, and let A be an attribute of R. The projection of R that removes the attribute A includes the usual data tuples, plus the knowledge tuples that have a blank (possibly suffixed by *) in attribute A.

In the example, assume attribute DEPARTMENT is removed from relation EMPLOYEE. Two knowledge tuples (C1 and I1) do not restrict this attribute and are, therefore, retained after the projection. The other knowledge tuples (C2 and I2) restrict this attribute (through the constant "Construction" and the variable v) and are, therefore, discarded.

Each knowledge tuple introduced in the above definitions must be shown valid, given the validity of the knowledge tuples in the input relations. Toward this end

we assume that the real world is captured in a hypothetical database; this database includes exactly the tuples that are true. A completeness assertion is a relational expression that, when computed in the database, derives a relation that *contains* the relation that would have been derived had the same expression been computed in the real world. Similarly, an integrity constraint is a relational expression that, when computed in *both* the database and the real world, derives relations that are null. With this hypothetical database, knowledge tuples that are introduced by relational operators may be given formal justifications. The proofs are quite similar, and we illustrate one example below.

Assume a completeness tuple r in relation R, and a completeness tuple s in relation S, and let T denote the Cartesian product of R and S. Let R', S', and T' be the corresponding relations in the real world. Consider the tuple rs. It specifies a subset of T and a subset of T'. Consider a tuple in the subset of T'. This tuple may be split into two separate tuples that are in R' and S', respectively. These tuples would be contained in the subsets specified by the completeness tuples r and s, respectively. Therefore, given the validity of the completeness tuples r and s, they are also in the subsets of R and S that are specified by r and s. Consequently, they must also be in the subset of T that is specified by rs. The conclusion is that rs is indeed a valid completeness tuple in the Cartesian product.

Often, a knowledge tuple that is retained after a selection operation may be "relaxed" slightly. For example, consider a knowledge tuple that has a constant a in attribute A and a selection condition $A = a$. This knowledge tuple is selected, but the semantics of the resulting relation permits us to clear this restriction in the new knowledge tuple. Similarly, consider a knowledge tuple that has the same variable x in both attributes A and B and a selection condition $A = B$. This knowledge tuple is selected, but the semantics of the result permit us to clear *at least one* of the restrictions in the new knowledge tuple (and both, if x does not appear anywhere else). The advantage of clearing such restrictions is that the new knowledge tuple will "survive" a future projection that removes this particular attribute.

We demonstrate these methods with the following query to retrieve the names of all the editors in Construction:

```
range of e is EMPLOYEE
range of s is SKILL
retrieve (e.E-NAME)
where e.E-NAME = s.E-NAME
  and s.S-NAME = "Editing"
  and e.DEPARTMENT = "Construction".
```

For clarity, this query is processed in nine steps:

(1) Perform the Cartesian product of EMPLOYEE and SKILL.
(2) Select from the result EMPLOYEE.E-NAME = SKILL.E-NAME.
(3) Remove from the result one of the attributes E-NAME.
(4) Remove from the result the attribute LEVEL.
(5) Select from the result S-NAME = "Editing".
(6) Remove from the result the attribute S-NAME.

(7) Select from the result DEPARTMENT = "Construction".
(8) Remove from the result the attribute DEPARTMENT.
(9) Remove from the result the attribute SALARY.

The first three steps correspond to a *natural join* between EMPLOYEE and SKILL, after which we have

	E-NAME	SALARY	DEPARTMENT	S-NAME	LEVEL
	Betty	38,000	Production	Cinematography	2
	John	26,000	Construction	Electrical	3
	John	26,000	Construction	Directing	1
	Tom	32,000	Marketing	Operating	2
	Mary	34,000	Production	Acting	2
	Mary	34,000	Production	Editing	3
C3	*		Construction	Electrical	
I3	*	z		Editing	
I4	*		v	Editing	

Note that whereas the set of all names of employees was complete (C1), the set of all names of employees with skills was not asserted to be complete. Consequently, the set of names of employees in this intermediate relation, which models only skilled employees, is not guaranteed to be complete. The fourth step removes the attribute LEVEL without any effect on the knowledge tuples. The fifth step now selects only one data tuple and two knowledge tuples, I3 and I4, where it clears the attribute S-NAME. The sixth step then projects out this attribute, yielding

	E-NAME	SALARY	DEPARTMENT
	Mary	34,000	Production
I5	*	z	
I6	*		v

The seventh step now removes the last data tuple and the knowledge tuple I5. In I6 it clears the attribute DEPARTMENT. The eighth step then projects out this attribute, and the last step projects out the attribute SALARY and removes I7. The result is

	E-NAME
I7	*

Since the final relation represents the set of all employees in Accounting who are editors, I7 is an integrity constraint that states that the set of all employees in Accounting who are editors is null (i.e., contained in one of the original null subsets). Since the answer includes no data tuples, the final verdict of the Supposition Tester is a double rejection: we know of no data to support the presupposition, and there may be no data to support the presupposition.

3.4 Performance Issues

As described in Section 2.4, the SEAVE mechanism simply combines SG (the supposition generalizor) with ST (the supposition tester) in a recursive procedure. The only additional issues we discuss here involve performance.

With the methods described above for storing and manipulating the knowledge, the cost of testing a presupposition is comparable to the cost of evaluating the query. However, as shown in Figure 2, to identify the erroneous presuppositions that are significant, numerous follow-up queries may need to be tested. This raises two important issues: (1) how far should the mechanism proceed in the analysis of a single query, and (2) are there any opportunities for optimization.

To prevent spending too much time on the analysis of a single failed query, it may be necessary to impose some kind of a limit, for example, a ceiling on the depth of the recursion, or a ceiling on the total number of presuppositions tested in connection with a single query, or simply a time limit. Here, we assume a limit of the first kind. Thus, the original query will be generalized in all possible directions, until it reaches a predetermined level.

By predetermining the farthest level of generalization, we can obtain at the beginning of the process a *cluster* of queries that eventually may have to be evaluated. Techniques for optimizing multiple queries have been examined in the past (e.g., [3], [10], [12], and [19]). However, the lattice structure among the queries in this cluster suggests a much simpler approach.

Let k be the predetermined depth of recursion. Given the input query, the techniques described in Section 3.1 are applied to weaken *each* of its subformulas by k steps. The resulting query will serve as the lower bound of a new lattice, which will include only the top k levels of the original lattice. In the example of Figure 2, assuming $k = 4$, the lower bound query is "all employees under 33 (of either sex) who earn at least \$37,000" (i.e., (33, *, 37)).

When a user query is received, it is first evaluated. If it fails, then the lower bound query is obtained, evaluated, and the resulting relation stored. Thereafter, the standard SEAVE algorithm is applied (to depth k). However, all follow-up queries are now evaluated on the basis of the result of the lower bound query. Thus, the cost of deriving the erroneous and significant presuppositions is reduced substantially.

4. CONCLUSION

Often, the source of null answers is in erroneous presuppositions that are undetected by the database management system. This article describes a mechanism called SEAVE that distinguishes between these fake nulls and genuine nulls. SEAVE attempts to verify the presuppositions behind queries by testing them against three repositories of database knowledge: the database itself, its completeness assertions, and its integrity constraints. When a query fails the test, it is repeated for generalizations of this query in an attempt to detect the erroneous presuppositions that are most significant.

This mechanism can be adapted for any database management system. In particular, we examined it in the environment of relational databases, where we showed a method for storing all three knowledge repositories together and a

procedure for testing queries against this knowledge, at a cost comparable to standard query evaluation.

Below are several examples of queries that failed and the reaction of SEAVE to each of them. For clarity, the queries and the reactions are presented in natural language. The phrases "there may not be . . . ," "we are certain there are no . . . ," and "we know of no . . . ," reflect the detection of erroneous and significant presuppositions based on, respectively, violation of an integrity constraint, containment in a complete subset, and a null answer.

1. Query: Retrieve all editors in Marketing who earn less than $28,000.
   SEAVE: There may not be editors in Marketing.
   There may not be editors who earn less than $30,000.
   We know of no employees in Marketing who earn less than $32,000.
2. Query: Retrieve all employees who earn more than $50,000.
   SEAVE: We are certain there are no employees who earn more than $38,000.
3. Query: Retrieve all level 1 operators in Production.
   SEAVE: We know of no operators in Production.
   We know of no level 1 operators.
   We know of no employees with level 1 skills in Production.

One weakness of the methods we have described is that currently they cannot combine completeness assertions with integrity constraints to infer new statements. For example, they do not conclude that the set of employees in Production is complete from the knowledge that the set of employees who know editing is complete and that every employee in Production is required to know editing.

The method of storing *generic* statements along with the facts has possibilities that are also outside the current scope of interpreting null answers. Techniques similar to those described in this article can be used to manipulate relations with stored constraints so that queries can be answered both specifically and generically. For example, if the constraint that every level 2 cinematographer earns at least $36,000 is stored in the relations, then the query "Who earns at least $35,000?" can now be given a more insightful answer: "Betty, Tom, and, in general, every level 2 cinematographer."

In this article we have described the principles and algorithms of a mechanism for detecting erroneous presuppositions in query systems. The implementation of this mechanism, in particular, the management of the database knowledge that we assumed, involves several engineering, as well as procedural, issues that have yet to be addressed—for example, how to assure the validity of the knowledge statements in a dynamically changing environment, and how to paraphrase and display knowledge statements to users.

REFERENCES

1. ARENS, Y. CLUSTER: An approach to contextual language understanding. Ph.D. dissertation, Dept. of Mathematics, Univ. of California, Berkeley, 1986.
2. BORGIDA, A. Language features for flexible handling of exceptions in information systems. *ACM Trans. Database Syst. 10*, 4 (Dec. 1985), 565–603.
3. CHAKRAVARTHY, U. S., AND MINKER, J. Processing multiple queries in database systems. *Database Eng. 1*, 1983.

4. CORELLA, F., KAPLAN, S. J., WIEDERHOLD, G., AND YESIL, L. Cooperative responses to Boolean queries. In *Proceedings of the 1st International Conference on Data Engineering* (Los Angeles, Calif.). IEEE Computer Society, Silver Spring, Md., pp. 77–85.
5. DATE, C. J. *An Introduction to Database Systems*, vol. 2. Addison-Wesley, Reading, Mass., 1983.
6. DATE, C. J. *An Introduction to Database Systems*, vol. 1, 4th ed. Addison-Wesley, Reading, Mass., 1986.
7. GALLAIRE, H., MINKER, J., AND NICOLAS, J.-M. Logic and databases: A deductive approach. *ACM Comput. Surv. 16*, 2 (June 1984), 153–185.
8. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351–386.
9. HENDRIX, G. G., SACERDOTI, E. D., SAGALOWICZ, D., AND SLOCUM, J. Developing a natural language interface to complex data. *ACM Trans. Database Syst. 3*, 2 (June 1978), 105–147.
10. JARKE, M. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Eds. Springer-Verlag, New York, 1984.
11. KAPLAN, J. Cooperative responses from a portable natural language data base query system. Ph.D. dissertation, Dept. of Computer and Information Science, Univ. of Pennsylvania, Philadelphia, 1979.
12. KIM, W. Global optimization of relational queries. In *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Eds. Springer-Verlag, New York, 1984.
13. LEBOWITZ, M. Generalization and memory in an integrated understanding system. Ph.D. dissertation, Dept. of Computer Science, Yale Univ., New Haven, Conn., 1980.
14. LEVESQUE, H. J., AND BRACHMAN, R. J. A fundamental tradeoff in knowledge representation and reasoning. In *Readings in Knowledge Representation*, R. J. Brachman, and H. J. Levesque, Eds. Morgan Kaufmann, Los Altos, Calif., 1985, pp. 42–70.
15. MOTRO, A. Query generalization: A method for interpreting null answers. In *Expert Database Systems (Proceedings from the 1st International Workshop)*, L. Kerschberg, Ed. Benjamin/Cummings, Menlo Park, Calif., 1986, pp. 597–616.
16. MOTRO, A. Completeness information and its application to query processing. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Aug. 25–28). Very Large Database Endowment, Morgan Kaufmann, Los Altos, Calif., 1986, pp. 170–178.
17. MOTRO, A. Supporting goal queries in relational databases. In *Proceedings of the 1st International Conference on Expert Database Systems* (Charleston, S.C., Apr. 1–4). Institute of Information Management, Technology and Policy, Univ. of South Carolina, Columbia, S.C., 1986, pp. 85–96.
18. MYLOPOULOS, J., AND WONG, H. K. T. Some features of the TAXIS data model. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Canada, Oct. 1–3). ACM, New York, 1980, pp. 399–410.
19. SELLIS, T. K. Global query optimization. In *Proceedings of ACM-SIGMOD International Conference on Management of Data* (Washington, D.C., May 28–30). ACM, New York, 1986, pp. 191–205.
20. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
21. ULLMAN, J. D. *Principles of Database Systems*. Computer Science Press, Rockville, Md., 1982.
22. WINOGRAD, T. *Understanding Natural Language*. Academic Press, Orlando, Fla., 1972.
23. ZLOOF, M. Query-by-example: A database language. *IBM Syst. J. 16*, 4 (Dec. 1977), 324–343.

Received March 1986; revised October 1986; accepted October 1986.