IDEA GROUP PUBLISHING

ITP4378

701 E. Chocolate Avenue, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

"Not" is Not "Not" Comparisons of Negation in SQL and Negation in Logic Programming

James D. Jones, Computer Science, College of Information Science and Systems Engineering
University of Arkansas at Little Rock, james.d.jones@acm.org, phone: 501-569-8138, fax: 501-569-8144

I. ABSTRACT

Logic programming presents an excellent paradigm within which to develop intelligent systems. In addition to the routine sorts of reasoning we would expect such a system to perform, the state of the art in logic programming allows one to reason explicitly about false information, to reason explicitly about unknown or uncertain information (the subject of this paper), and to reason introspectively in spite of holding multiple competing ways of viewing the world. Each of these abilities are necessary forms of reasoning, and each are lacking from current technology. Needless to say, each of these abilities are also lacking from relational databases. In this paper, the focus is upon the expressive power of weak negation in logic programming. Weak negation is not well understood, and can be easily confused with negation in SQL. In particular, some may falsely equate weak negation with the "not exists" clause of SQL. It is true that both forms of negation do have some similarity. However, the expressive power of weak negation far exceeds that of "not exists" in SQL. To a much lesser extent, we shall also consider strong negation in logic programming.

II. INTRODUCTION

As we consider a future of very intelligent machines, we need enabling technologies that will allow us to represent and reason about all forms of knowledge. Those enabling technologies will concern themselves not only with hardware improvements, but much more importantly, with software improvements. We could have the fastest chips, busses, and memory conceivable, and we could have the most capacious memory and storage medium, yet without very significant improvements to our reasoning methods, all we gain is faster turnaround time and greater throughput. We need processing abilities, reasoning abilities, that are orders of magnitude beyond current technology.

The software improvements we need are not concerned so much with programming languages and operating systems, important as these are. Rather, we are concerned with the essence of thought. How do we get an inanimate object to perform such thought? The quest of the artificial intelligence community at large is to produce a self-sufficient reasoning machine much like the computer 'HAL' in the movies "2001: A Space Odyssey", and "2010: The Year We Make Contact". A more contemporary example is the robot in the movie "BiCentennial Man".

While others exalt the nature of such a successful inanimate object (Moravec 1999), the bottom line is that we have to devise ways to get something akin to a doorstop to perform magnificent feats that we call thought. This is incredibly difficult. We need strategies or methods to perform very difficult forms of reasoning. We also need these strategies to be on solid theoretical foundations. This is the goal of the declarative semantics of logic programming. The quip is that we need to reason in semantically correct ways.

Reasoning methods are only part of the requirement for deep reasoning. Another part of the picture is that we need knowledge. A successful machine reasoner must posses both: a large quantity of knowledge, and the ability to use that knowledge to create new knowledge (make inference, make decisions, etc.) Traditionally, databases have been viewed as the repository of vast amounts of information, while logic programming has been viewed as an inference engine to allow us to derive new information from existing information. In this paper, we are concerned about negation in these fields, since: 1) these fields are related, and 2) we desire to provide a greater appreciation for the power of logic programming. The overlap of these two fields, that is, intelligent databases, has come to be known as deductive databases.

The next section primarily discusses what weak negation means. In cursory manner, it also discusses strong negation. Strong negation in logic programming is something SQL does not have an analogue for. The subsequent section illustrates by way of examples how weak negation is far more powerful than negation in SQL. While formal proofs are beyond the scope of this paper, it should be apparent that weak negation completely subsumes negation in SQL, and far exceeds the expressive power of negation in SQL. It is assumed that the reader has a working knowledge of SQL. (For those readers not very familiar with SQL, (Connolly, Begg 1999) provides an excellent and very readable discussion of SQL.)

Throughout, reference will be made to relational databases. From the point of view of semantics, relational databases have a very strong mathematical foundation, and therefore a good subject for us to consider. From this standpoint, the reader should not view relational databases as passe', preferring object oriented databases or multimedia databases instead. For the purposes of this discussion, from the view of semantics, object oriented databases and multimedia databases do not offer greater expressive power. This is not to say that these advanced databases are not very useful, and this is not to say that these databases do not allow us to incorporate new types of information. This is to say that semantically speaking, there is not a difference between pointing to a scalar value and pointing to an X-ray in terms of abstract computational complexity.

III. WEAK NEGATION IN LOGIC PROGRAMMING

Negation is quite an important topic in logic programming (Apt, Bol 1994). From the point of view of semantics, there are two primary forms of negation in logic programming. One form of

This paper appears in *Managing Information Technology in a Global Economy*, the proceedings of the Information Resources Management Association International Conference.

negation is strong negation (Gelfond, Lifschitz 1990), or classical negation, and is designated by the operator¬. This form of negation means that something is definitively false. An example would be something such as

¬*angry*(*john*).

which means that it is a fact (with respect to our view of this world) that *john* is not angry. Now such a fact may be explicitly stated, or it may be inferred. Nonetheless, with respect to the beliefs of our program, it is definitively believed that *john* is not angry.

The other form of negation is weak negation (or negation-as-failure) (Lifschitz 89). This form of negation is designated by the operator *not* which means that a fact is not known (or more precisely, not provable.) For instance,

not angry(*john*)

means it cannot be proven that *john* is angry. That is, with respect to our view of the world, it is not currently believed that *john* is angry. In fact, *john* may be angry, or *john* may not be angry, however, with respect to our knowledge (as embodied by our database or logic program), neither of these facts can be determined. (Actually, this statement is too strong, and a more correct statement is beyond this paper. However, the intuitive meaning of this statement can be easily implemented) The first example states that it is definite that *john* is not angry; the second example states that it is not believed that *john* is angry. The second is a far weaker statement about *john*'s anger. In the first example, there is presence of information that causes us to hold this belief; in the second example, there is absence of information that causes us to hold this belief. This is a vital distinction.

It is weak negation that provides the ability to represent and reason explicitly about unknown or uncertain information. It is important to note that weak negation is part of an inference mechanism, and is not at all mutually exclusive with or in competition with other forms of uncertain reasoning such as fuzzy sets, certainty factors, neural nets, or probability theory. Further, not only can we represent the fact that there is absence of information, but we can also use such a fact to infer new information. Consider the following:

safe(*sue*) ¬ *not angry*(*john*)

which states that *sue* is safe if it is not known that *john* is angry. (Perhaps *sue* knows that if *john* were angry, she would know about it, because *john* is so transparent.)

The confusion

These forms of negation in logic programming, and in particular weak negation, can easily be confused with negation in SQL. It is easy to see how such a confusion can arise. First of all, there is a syntactic similarity between weak negation's operator *not* and SQL's *not exists*. Secondly, there is an intuitive similarity in that both express the fact the some information is missing. To illustrate these two reasons for the confusion, let us consider the following example.

Example 1

Assume that the following is a database for a company that places its employees on contract with other firms.

EMPLOYEE

Name	*Skill*	*Availability*
John	carpenter	full-time
Jay	plumber	full-time
Mark	manager	full-time
Sally	accountant	part-time

CURRENTLY ASSIGNED

Name	*Contracting Org.*	*Hrs. of Contract Remaining*
John	Alltel	500 hours
Sally	RCA	120 hours

The EMPLOYEE relation lists all employees. The CURRENTLY_ASSIGNED relation lists those employees that are already out on contract. These relations could be equivalently expressed as a logic program, as in the following:

employee(*john, carpenter, full-time*).
employee(*jay, plumber, full-time*).
employee(*mark, manager, full-time*).
employee(*sally, accountant, part-time*).
currently_assigned(*john, alltel, 500*).
currently_assigned(*sally, rca, 120*).

Suppose we are interested in identifying those employees that are not yet contracted out. In SQL, such a request would be satisfied by the query:

```
SELECT *
FROM employee
WHERE NOT EXISTS
    (SELECT *
     FROM currently_assigned
     WHERE employee.name = currently_assigned.name);
```

This query will produce a report on which only *jay* and *mark* appear. The exact same information could be gleaned from a logic program using the following rule:

available_for_work(*Name,Skill,Availability*) ←
    employee(*Name, Skill, Availability*),
not currently_assigned(*Name, Contracting_org,*
    Hours_remaining).

This rule states that those employees available for work (contract) are those not currently assigned.

In this example, the two paradigms produce exactly the same results. It is easy to see the syntactic similarity: SQL uses the form *NOT EXISTS* (*... currently_assigned*), and logic programming uses the form *not currently_assigned*. In both cases, *not* appears. It is also easy to see the intuitive similarity: in both cases we trigger on the fact that something is missing. As we will see in the next section, the similarity ends here.

IV. COMPARISON OF WEAK NEGATION, AND NEGATION IN SQL

Those who are not intimately informed of the semantics of logic programming may fail to see the additional expressive power that logic programming provides over SQL. This section focuses only on one aspect of logic programming, weak negation. The purpose here is to examine by way of examples negation in each of these two paradigms to demonstrate the greater expressive power that weak negation provides.

Example 2

Let us return to the example 1 of the previous section. The example, as stated, produces the exact same results as the SQL query. That example uses the logic programming rule:

available_for_work(*Name, Skill, Availability*) ←
    employee(*Name, Skill, Availability*),
not currently_assigned(*Name, Contracting_org, Hours_remaining*).

Admittedly, this rule matches our intuition, and it produces the desired results. Both paradigms, SQL and logic programming yield that same results, that is that *jay* and *mark* are available for work. From this point on, the similarities between SQL and logic programming cease. The remainder of this example, and the other examples exceed the expressive power of SQL.

If we dropped the goal *employee*(*X, Y, Z*), then we would have the following rule:

available_for_work(*Name, Skill, Availability*) ←
not currently_assigned(*Name, Contracting_org, Hours_remaining*).

This rule is a much more powerful rule and would have yielded all objects in our Herbrand interpretation which/who are not currently assigned. Consider for instance, that we also had the following relation:

FORMER_EMPLOYEE

Name
Rachel
Suzzie
Bob

Then in addition to *jay* and *mark*, this rule would also infer that *rachel*, *suzzie*, and *bob* are available to work. The additional inferences that *rachel*, *suzzie*, and *bob* are also available to work are beyond the expressive ability of SQL. There may be applications where ascribing a property or a relation to the rest of the objects of the Herbrand interpretation is appropriate. •

Unfortunately, in this example, the rule as it stands would also infer that *carpenter*, *plumber*, *part-time*, etc., are available for work, because these are also objects in our Herbrand interpretation. However, appropriate constraints could be placed upon the rule so that we do get the results we desire. The point is not that the above rule yields counter-intuitive results; the point is that given the same data, logic programs have the ability to infer far more than what SQL will produce for us. As already mentioned, there may be applications where this unthrottled approach is suitable. Yet, we also have the ability to restrict our inferences to get the same results as does SQL.

It is very important to note that with logic programming, we have the power of inference. By contrast, with SQL, we have static reporting capability. Inference is dynamic. In a sense, it updates our database. So regardless which version of the logic programming rule we use (the rule from example 1, or the rule from example 2), the result is a more informed database.

It is also important to note that in some respect, we have inferred something from nothing. (There is not truly nothing, because the definition of a Herbrand interpretation identifies the object constants in our language.) In example 1, SQL had success in producing results by referring to an existing relation, EMPLOYEE. By contrast, the logic programming rule introduced in example 2 yielded names that exist in the FORMER_EMPLOYEE relation without ever referring to that relation.

A very powerful feature of weak negation is that it allows us to express uncertainty in the sense that we may express that one or more of several alternatives may be true. This ability induces several competing views of the world. Yet, we still have the ability to reason in spite of these competing views, as demonstrated in this next example.

Example 3

Let us return to the relations from example 1. Assume that we are recruiting a new-hire for a position, and we have narrowed the field down to two candidates: *tom* and *victoria*. That is, at this moment, we know that we will hire either *tom* or *victoria*, but we do not yet know which. Therefore, with respect to this example, we have two competing views of the world: one in which *tom* will be hired, and another in which *victoria* will be hired. Let us lay aside the additional complexities which a temporal database would have (that is, the ability to reason across multiple views of time), and consider that whichever candidate we will hire is available for work (at some time in the future). We have two candidates for what our relations would look like. One view of our world indicates that we will hire *tom*, represented by the following relations.

EMPLOYEE

Name	*Skill*	*Availability*
John	carpenter	full-time
Jay	plumber	full-time
Mark	manager	full-time
Sally	accountant	part-time
Tom	doctor	full-time

CURRENTLY ASSIGNED

Name	*Contracting Org.*	*Hrs. of Contract Remaining*
John	Alltel	500 hours
Sally	RCA	120 hours

Given this view of the world (with respect to this example), the same SQL query or the same logic programming rule from example 1 would inform us that *jay*, *mark*, and *tom* have not been assigned to a contract, and are available for work.

The other view of our world indicates that we will hire *victoria*. The relations in this view of the world are represented by the following.

EMPLOYEE

Name	*Skill*	*Availability*
John	carpenter	full-time
Jay	plumber	full-time
Mark	manager	full-time
Sally	accountant	part-time
Victoria	doctor	full-time

CURRENTLY ASSIGNED

Name	*Contracting Org.*	*Hrs. of Contract Remaining*
John	Alltel	500 hours
Sally	RCA	120 hours

Given this view of the world, the same SQL query or the same logic programming rule from example 1 would inform us that *jay*, *mark*, and *victoria* have not been assigned to a contract, and are available for work.

Clearly, the results of these two queries are different. Relational database technology does not have the capacity to represent these two competing ways of viewing the world. Certainly, if we cannot represent it, we cannot query against it with SQL. By contrast, this information is very easily represented by the following logic program.

employee(*john, carpenter, full-time*).
employee(*jay, plumber, full-time*).
employee(*mark, manager, full-time*).
employee(*sally, accountant, part-time*).
currently_assigned(*john, alltel, 500*).
currently_assigned(*sally, rca, 120*).
employee(*tom, doctor, full-time*) ←
  not employee(*victoria, doctor, full-time*)
employee(*victoria, doctor, full-time*) ←
  not employee(*tom, doctor, full-time*)

This logic program has two belief sets: one in which *tom* is considered an employee, and one in which *victoria* is considered an employee. (Note that in terms of logical entailment, the order of the rules is completely unimportant. The new information has been added to the bottom to make it easier to identify the differences between the examples.)

In addition to being able to represent and reason about different views of the world, logic programs can also reason about sets of items (Beeri, et.al 1991; Dovier, et. al. 1996; Jones 1999). That is, a set can be a term, an object of the universe of discourse. A term is akin to a single cell in a relation. (A single cell being the intersection of a tuple and an attribute.) The concept of a set of values to be taken as the current value of a cell is totally lacking from relational technology. Consider the following example.

Example 4

Let us consider who is on the payroll at a particular company. This company has a basketball team that it supports as a publicity aid. Since the composition of the team constantly changes (personnel turnover, the total number of players fluctuates, etc.), the company has adopted the practice that it writes one check to the team as a whole, and the team distributes those proceeds as it sees fit. So, those on the payroll include the employees of the company, and the basketball team (which is treated as a single entity.)

The following represents the basketball team:

basketball_player(*tony*).
basketball_player(*hakim*).
basketball_player(*clyde*).
basketball_player(*michael*).
basketball_player(*stuart*).
basketball_player(*henry*).

The following rules (appealing to the information from example 1) represents who is on payroll.

on_payroll(*Name*) ← *employee*(*Name, Skill, Availability*).
on_payroll(*Set*) ← *setof*(*Name, basketball_player*(*Name*), *Set*).

The intensional relation *on_payroll* is the following.

on_payroll(*john*).
on_payroll(*jay*).
on_payroll(*mark*).
on_payroll(*sally*).
on_payroll({*tony, hakim, clyde, michael, stuart, henry*}).

We see that there are five entities on payroll: *john, jay, mark, sally*, and the set of players {*tony, hakim, clyde, michael, stuart, henry*}.

This ability to represent and reason about collections of objects is very important. Further, the ability define such collections intensionally (that is, by a somewhat mathematical definition rather than explicitly listing the collection) is very powerful. Allowing the intensional definition to use the full power of logic programming formulae, including weak negation, is powerful indeed. The following example shows the use of constructing such a set using weak negation.

Example 5

Continuing from the previous example, let us assume that this same company also provides athletic scholarships to the local university. The company has the stipulation that recipients of the athletic scholarship must also play for the company team. However, since the individual is receiving a scholarship, the individual is not to receive compensation from the team. In this strange world that we are constructing, this does not violate NCAA rules. The following rule represents the fact that *henry* is on scholarship.

on_scholarship(*henry*).

Now let us rewrite the rules for defining who is on payroll.

on_payroll(*Name*) ← *employee*(*Name, Skill, Availability*).
on_payroll(*Set*) ← *setof*(*Name,*(*basketball_player*(*Name*), *not on_scholarship*(*Name*)), *Set*).

Now the intensional relation *on_payroll* is the following.

on_payroll(*john*).
on_payroll(*jay*).
on_payroll(*mark*).
on_payroll(*sally*).
on_payroll({*tony, hakim, clyde, michael, stuart*}).

In the previous example, *henry* was among the set of basketball players considered on payroll. In this example, *henry* is not among the set of basketball players considered on payroll. This difference has been achieved by the use of weak negation in the intensional set definition, all of which is not achievable in relational databases and SQL.

IV. SUMMARY AND FUTURE WORK

This paper has examined the simplest form of weak negation: that which appears in extended logic programs. Even extended logic programs are beyond current practice in terms of expressive power. There are yet more powerful semantics for logic programs which introduce additional opportunities for weak negation to represent different classes of problems. This present work could be extended to consider those more powerful languages.

In comparing weak negation with negation in SQL, we have seen by example that weak negation can represent the same information as that represented by SQL. We have also seen several other examples where weak negation can represent problems not able to be expressed in relational databases. These examples include: the fact that in its simplest form, weak negation can perform

more inferences than SQL (appealing to ground terms in the Herbrand interpretation); the fact that weak negation allows us to represent multiple views of the world; the fact that extended logic programs (which are the minimum platform for weak negation) can be extended to allow representation of sets (extensional and intensionally); and the fact that weak negation can be used in those intensionally set definitions.

There are other ways that this present work can be expanded. Future work can recast the present work into a more formal work with proofs. Considering a totally different matter, a related concept to negation is missing information. Future work could examine the relationship between null values in database technology and weak negation. Further, SQL does allow nesting of subqueries. (Such would be of the form of a subquery that uses the Anot exists@ clause.) Perhaps the limits of nesting should be considered. Very much related to this is the idea that SQL deals with only one relation at a time, and deals sequentially and deterministically with multiple relations. In logic programming, multiple relations can easily be dealt with in all possible combinations of Horn clauses, strong negation and weak negation, and order does not matter (to semantics.) Further, parallelism does not affect the semantics. The comparison between database technology and logic programming could be continued along these lines of nesting, and considering multiple relations nondeterministically.

REFERENCES

(Apt, Bol 1994) Apt, Krzysztof R., and Roland N. Bol: Logic Programming and Negation: A Survey, *Journal of Logic Programming*, vol 19/20 May/July 1994.

(Beeri, et. al. 1991) Beeri, S. Naqvi, O. Shmueli, and S. Tsur: Set Constructors in a Logic Database Language, *Journal of Logic Programming*, 10(3):181-232, 1991.

(Connolly, Begg 1999) Connolly, Thomas, and Carolyn Begg: *Database Systems, A Practical Approach to Design, Implementation, and Management*, 2nd Ed., Addison-Wesley, 1999

(Dovier et. al. 1996) Dovier, Agostino, Enrico Pontelli, and Gianfranco Rossi: {log}: A language for programming in logic with finite sets, *Journal of Logic Programming*, 28(1):1-44, 1996.

(Gelfond, Lifschitz 1990) Gelfond, Michael, and Vladimir Lifschitz: Logic Programs with Classical Negation. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proceedings or the 7th Int=l Conf*, 1990.

(Lifschitz, 1989) Lifschitz, Vladimir: Logical Foundations of Deductive Databases, *Information Processing* 89, North-Holland

(Jones 1999) Jones, James D.: AA Declarative Semantics For Sets Based on the Stable Model Semantics@, *Declarative Programming with Sets* (*DPS >99*), in conjunction with *Principles, Logics, and Implementations of high-level programming languages* , Paris, France, Paris, France, 1999

(Moravec 1999) Moravec, Hans: *ROBOT, Mere Machine to Transcendent Mind*, New York, NY, : Oxford University Press, 1999