1From Semantic Representations to SQL Queries

Per Anker Jensen, Bodil Nistrup Madsen
Annie Stahél, Carl Vikner
København

Abstract

Our paper discusses problems which arise when trying to translate a semantic representation into a SQL database query, and more particularly the encoding of yes/no-questions, and the evaluation of the existential and the universal quantifier.

1 The project

Our project investigates the problems which arise in creating a natural language interface to the database query language SQL. The basic layout of our system is a stepwise progression from a natural language expression via its semantic representation to a SQL query. The reason for choosing SQL as the database query language is that it is widely used in conventional database systems, like ORACLE for instance.

In our talk, we will be concerned specifically with the problems which arise when trying to translate a semantic representation into a SQL query, and more particularly with three problems which are of special importance for a natural language interface, namely the encoding of yes/no-questions, and the evaluation of the existential and the universal quantifier.

2 Semantic representations and restricted quantification

The semantic representation we employ is in the form of restricted quantification (cf. Jensen & Vikner (1992, I: 137–48)). An example with one quantifier is shown in (1):

(1) exists(x,customer1a(x),complain1a(x))
'En kunde klager'
'A customer complains'

Formulas like this one consist of four components as indicated in (2):

(2)

exists	(x,	customer1a(x),	complain1a(x))
↑	↑	↑	↑
QUANT	VAR	RESTRICTION	ASSERTION

Proceedings of NODALIDA 1993, pages 133-142

A formula with the format shown in (2) we call a 'quantifier structure'. By 'predicate structure' we refer to expressions such as those which make up the restriction and the assertion in (2). These consist simply of a predicate followed by a number of arguments depending on the arity of the predicate. In this example, the meaning of the head noun of the subject, i.e. *kunde* ('customer'), is represented as the predicate structure in the restriction slot of the formula, and the intransitive VP *klager* ('complains') is represented as the predicate structure in the assertion slot.

It should be mentioned that neither the restriction nor the assertion is necessarily a predicate structure. Rather, they may contain any well-formed formula, hence also quantifier structures. An example of this is shown in (3), which contains a quantifier structure in the assertion slot.

(3) exists(y,product1a(y),
all(x,customer1a(x),complain2a(x,y))
'Alle kunder klager over en vare'
'All customers complain of a product'

The point of using restricted rather than unrestricted quantification, as is customary in classical predicate logic, is that when using restricted quantification, only a subset of the individuals in the domain is quantified over, namely those individuals which satisfy the restriction. This comes out clearly in the evaluation algorithms we propose for the quantifiers.

We want the evaluation of the quantifiers *exists* and *all* to be taken care of by the rough algorithms in (4) and (5), respectively, which ensure that only the individuals satisfying the restriction are considered when evaluating the assertion.

(4) **Evaluation of formulas of the form: exists(x,restriction(x),assertion(x))**

1. Find all the individuals that satisfy the restriction
2. IF at least one of the individuals found in step 1 satisfies the assertion
   THEN the formula evaluates to true
   ELSE the formula evaluates to false.

(5) **Evaluation of formulas of the form: all(x,restriction(x),assertion(x))**

1. Find all the individuals that satisfy the restriction
2. IF all the individuals found in step 1 satisfy the assertion
   THEN the formula evaluates to true
   ELSE the formula evaluates to false.

3 Database tables and semantic predicates

For the purpose of this paper we have designed a small toy database, whose tables are shown in (6). The database contains the names of four customers and three types of products and, in the table *cp*, are registered the complaints made by some of the customers.

(6) Tables in the database:

c (customers)

NO	NME
1	Hansen
2	Jensen
3	Madsen
4	Sørensen

p (products)

NO	TYP
1	television set
2	video recorder
3	video camera

cp (complaints)

NO	CNO	PNO
1	3	1
2	2	1
3	3	2
4	1	1

When we want to evaluate a semantic representation with respect to this database, we have to relate the predicates of the semantic representation to the tables in the database. Following Grosz et al. (1987:222), this is done by means of a set of definitions of the predicates in terms of database tables as shown in (7).

(7) Definitions of semantic predicates in terms of database tables:

```
complain1a(Nme) ←
        c(Cno,Nme),
        cp(_,Cno,_)

complain2a(Nme,Pno) ←
        c(Cno,Nme),
        cp(_,Cno,Pno)
```

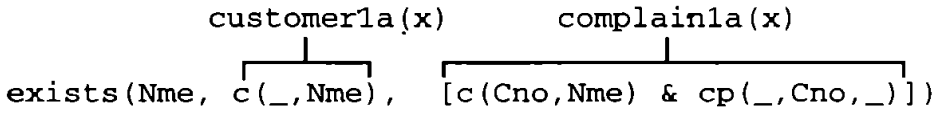
customer1a(Nme) ←
    c(_,Nme)

product1a(Pno) ←
    p(Pno)

television_set1a(Pno) ←
    p(Pno,'television set')

The predicate structures *customer1a(x)* and *complain1a(x)* in the semantic representation in (2) can now be replaced by their respective definitions. By doing so we obtain the expression shown in (8), which we call the tabular representation:

(8)



4 SQL and yes/no-questions

The SQL language[1] offers a facility for retrieving information from a database, namely the so-called SELECT queries. SELECT queries come in two types. A set-valued type and a number-valued type.

(9)

```
SELECT * FROM...          ┐
                          ├ set-valued type
SELECT c.NME FROM...      ┘

SELECT COUNT(*) FROM...      number-valued type
```

In the set-valued type, the SELECT list, i.e. the expression between the token *SELECT* and the token *FROM*, is a sequence of column specifications. The value of this type of queries is the set of tuples of values in the indicated columns of the rows which satisfy the condition of the query. To represent the value of such a query one can use the set notation proposed by Pirotte (1978:414), as shown in (10):

(10) { (x,y) | t(x,y) }

[1]For details of the SQL language, see for instance Date (1990), Ørum (1990) or *SQL Language Reference Manual. ORACLE* (1990).

In this notation the SELECT list *(x,y)* is written in front of a tabular formula containing the corresponding free variables. If we have a query of the form (11):

(11) SELECT c.NO, c.NME FROM c;

we can represent its value by means of the set expression in (12):

(12) { (NO,NME) | c(NO,NME) }

In the kind of number-valued type which is relevant to the subject of this paper, the SELECT list consists of the expression *COUNT(*)*. The value of a query of this form is the number of rows in the table which satisfy the condition of the query.

Numbers and sets (of tuples) are the only two kinds of possible answers to SQL queries. That is, unlike for instance Prolog, SQL does not support yes/no-questions directly. Therefore, we have to somehow trick it into doing so. Our stratagem consists in making use of the built-in SQL table DUAL. DUAL is a table with one column and one row with the value *X*. We begin all queries which encode a yes/no-question by the expression *SELECT COUNT(*) FROM DUAL*. Such a query yields the answer *1* if the condition is satisfied, and *0* otherwise. So, this gives us the equivalent of a yes/no-question facility.

The content proper of the yes/no-question is encoded by means of a SELECT subquery. This is shown in example (13), where the content 'Hansen complains' is encoded in the condition in the innermost WHERE clause, which checks the occurrence of a customer name *Hansen* whose customer number appears in a row in the complaints table. This SELECT subquery is made part of the WHERE clause of the outermost SELECT statement by means of the operator EXISTS. In this way we get a condition which comes out true – and thus triggers a *1* as the final answer – only in the case where the value of the SELECT subquery is nonempty.

(13)

a. *Hansen klager*
'Hansen complains'

b. Semantic representation:
complain1a(Hansen')

c. Tabular representation:
c(Cno,'Hansen') & cp(_,Cno,_)

d. SQL query:

```
SELECT COUNT(*) FROM DUAL
WHERE EXISTS
  (SELECT * FROM c, cp
  WHERE c.NME = 'Hansen'
  AND c.NO = cp.CNO);
```

5 Existential quantification

Turning next to existential quantification, Pirotte (1978: 419) has it that one can transform a formula containing the existential quantifier into an equivalent set expression, and thus remove the existential quantifier.[1] For instance (14.a) can be transformed into (14.b):

(14) a. exists(x,p(x),q(x))
b. { x | p(x) & q(x) } ≠ ∅

We use a transformation like the one in (14) as the basis for translating existentially quantified formulas into SQL queries. Thus the existentially quantified semantic representation in example (15.b and c) is transformed into the SELECT subquery in example (15.d) which encodes a set expression corresponding to the lefthand side of (14.b).

(15)

a. *En kunde klager*
'A customer complains'

b. Semantic representation:

```
exists(x,customer1a(x),complain1a(x))
```

c. Tabular representation:

```
exists(Nme,c(_,Nme),
                [c(Cno,Nme) & cp(_,Cno,_)])
```

d. SQL query:

```
SELECT COUNT(*) FROM DUAL
WHERE EXISTS
      (SELECT c.NME FROM c, cp
      WHERE c.NME LIKE '%'
      AND c.NO = cp.CNO);
```

[1]For other discussions of the elimination of the existential quantifier in database queries, see e.g. Minker (1978: 110), Dilger & Zifonun (1978: 395-400), Pereira (1983: 21), Steiner (1988: 186-87).

The LIKE-condition of the inner WHERE clause is redundant and is deleted by optimization.

Note that the *EXISTS* of the outer WHERE clause corresponds, not to the existential quantifier, but to the symbols "$\neq \varnothing$" of (14.b). So, in our treatment the existential quantifier disappears altogether. However, it would be possible to encode the quantifier *exists* by the SQL-operator *EXISTS*. In example (15) this would give a SQL query identical to the one shown. But in more complicated cases, i.e. examples containing multiple occurrences of quantifiers, this would result in a considerable number of subqueries (cf. Madsen & Stahél (forthcoming)).

6 Universal quantification

For the encoding of universal quantification we use the SQL operator MINUS, as shown in example (16). MINUS takes as its arguments two SELECT queries of the set-valued type and maps them onto the set-theoretic difference between the value of the first and the value of the second. The idea is that if we want to find out if all customers complain of some product, as in example (16), we find the difference between the set of customers and the set of complainers. If the resulting set is empty, then all customers complain, and so the initial query should receive a positive answer. That is, in the case of universal quantification, the subquery is a MINUS construction, and the value of this subquery must be the empty set for the outermost SELECT query to yield the value *1*. That is why the condition of the outermost WHERE clause is constructed by means of the expression *NOT EXISTS*.

The MINUS operator must be given comparable sets as arguments. In our example (16) these are sets of customer names determined by the SELECT list *c.NME* figuring in both SELECT expressions. This column designation is the encoding of the variable bound by the universal quantifier in the semantic representation.

(16)

a. *Alle kunder klager over en vare*
'All customers complain of a product'
= 'Each customer complains of some product'

b. Semantic representation:
all(x,customer1a(x),
exists(y,product1a(y),complain2a(x,y)))

c. Tabular representation:

```
all(Nme,c(_,Nme),
        exists(Pno,p(Pno,_),
                [c(Cno,Nme) & cp(_,Cno,Pno)]))
```

d. SQL query:

```
SELECT COUNT(*) FROM DUAL
WHERE NOT EXISTS
        (SELECT c.NME FROM c
        MINUS
        SELECT c.NME FROM p, c, cp
        WHERE p.NO = cp.PNO
        AND c.NO = cp.CNO);
```

The MINUS solution to universal quantification is analoguous to the analysis advocated by the theory of generalized quantifiers, which states the truth conditions of an expression of the form in (17):

(17) all N VP

as shown in (18):

(18) $[\![\text{all N VP}]\!] \equiv [\![\text{N}]\!] \subseteq [\![\text{VP}]\!]$

(cf. Barwise & Cooper (1981: 169), Thomsen (forthcoming)). The subset statement in (18) is equivalent to a statement in terms of set-theoretic difference of the form given in (19):

(19) $[\![\text{N}]\!] - [\![\text{VP}]\!] = \varnothing$

And this again is exactly what we have encoded by means of the MINUS construction.

Until this point the encoding of existential quantification has been relatively easy, and we have avoided the burden of keeping track of variables bound by the existential quantifier. However, if we have a semantic representation with a universal quantifier in the scope of an existential quantifier, as in example (20), such recklessness is no longer admissible. In example (20), the existential quantifier binds the variable designating the product, i.e. *y* or *Pno*. This variable appears again inside the scope of the universal quantifier. The point is that, for the formula to be true, it must be possible to find one particular value for this variable such that all customers complain of the product which has this number. Therefore we have to fix values for the variable outside the scope of the universal quantifier. This is done by giving the existential SELECT subquery in example (20) the SELECT list *p.NO* and repeating this

column specification in the universal SELECT subquery in the last AND clause, where it is required to be identical to *cp.PNO*.

(20)

a. *Alle kunder klager over en vare*
'All customers complain of a product'
= 'There is a product which all customers complain of'

b. Semantic representation:
exists(**y**,product1a(y),
all(x,customer1a(x),complain2a(x,**y**)))

c. Tabular representation:
exists(**Pno**,p(Pno,_),
all(Nme,c(_,Nme),
[c(Cno,Nme) & cp(_,Cno,**Pno**)]))

d. SQL query:

```
SELECT COUNT(*) FROM DUAL
WHERE EXISTS
  (SELECTS p.NO FROM p
   WHERE NOT EXISTS
     (SELECTS c.NME FROM c,
   MINUS
     SELECT c.NME FROM c,
   WHERE c.NO = cp.CNO
   AND p.NO = cp.PNO));
```

universal subquery

existential subquery

Thus, only in cases like this one, where the existential quantifier has a universal quantifier in its scope, do we have to keep track of an existentially bound variable.

7 Conclusion

To summarize, the table DUAL is used to encode yes/no-questions, the existential quantifier may be eliminated and the universal quantifier is encoded by means of the MINUS operator.

References

Barwise, Jon and Robin Cooper. 1981. *Generalized Quantifiers and Natural Language.* LINGUISTICS AND PHILOSOPHY, 4, 159–219.

Date, C.J. 1990. *An Introduction to Database Systems.* Volume I, Fifth Edition, Addison–Wesley, Reading, Massachusetts.

Dilger, Werner and Gisela Zifonun.1978. *The Predicate Calculus-Language KS as a Query Language.* In Gallaire and Minker, pp. 377–408.

Gallaire, Hervé and Jack Minker (eds). 1978. *Logic and Databases*, Plenum Press, New York.

Grosz, Barbara J., Douglas E. Appelt, Paul A. Martin and Fernando C.N. Pereira. 1987. *TEAM: An Experiment in the Design of Transportable Natural-Language Interfaces.* ARTIFICIAL INTELLIGENCE 32, 173–243.

Jensen, Per Anker and Carl Vikner. 1992. *Natursprogsbehandling og unifikations-grammatik*, I–II. Department of Computational Linguistics, Copenhagen Business School.

Madsen, Bodil Nistrup and Annie Stahél (forthcoming). *Database structure and SQL-queries for a natural language interface project.* LAMBDA 19, 1993.

Minker, Jack. 1978. *An Experimental Relational Data Base System Based on Logic.* In Gallaire and Minker, pp. 107–21.

Pereira, Fernando. 1983. *Logic for Natural Language Analysis.* Technical Note 275, SRI International, Menlo Park, California.

Pirotte, Alain. 1978. *High Level Data Base Query Languages.* In Gallaire and Minker, pp. 409–36.

SQL Language Reference Manual. ORACLE. 1990. Oracle Corporation.

Steiner, Juraj. 1988. *Relational theory of queries.* DATA AND KNOWLEDGE ENGINEERING 3, pp. 181–96.

Thomsen, Hanne Erdman (forthcoming). *The Semantic Values of NPs.* LAMBDA 19.

Ørum, Henning. 1990. *Oracle-håndbogen.* Borgen, Copenhagen.