Masque/sql– An Efficient and Portable Natural Language Query Interface for Relational Databases*

I. Androutsopoulos G. Ritchie P. Thanisch

Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, Scotland, U.K.
email: ion@aisb.ed.ac.uk, graeme@aisb.ed.ac.uk, pt@dcs.ed.ac.uk

1 Abstract

Masque is a powerful and portable natural language front-end for Prolog databases. It answers written English questions by generating Prolog queries, which are evaluated against the Prolog database. This paper describes a modified version of Masque, called Masque/sql, which answers English questions by generating and executing SQL code. Masque/sql maintains the full linguistic coverage of the original Masque, and can be used with any database system supporting SQL. Masque/sql is efficient, and can be easily configured using the built-in domain-editor.

2 Introduction

Masque (Modular Answering System for QUeries in English) is a natural language query interface for databases, developed at the Artificial Intelligence Applications Institute and the Department of Artificial Intelligence of the University of Edinburgh. The system, a descendant of Warren and Pereira's CHAT-80 [1], transforms written English questions into Prolog queries, which are executed against the Prolog database. Masque combines extensive linguistic coverage, efficiency and portability. Complex English questions are answered in a couple of seconds, and in cases of questions that cannot be answered cooperative messages are generated. The system is user-friendly, and can be easily configured for new knowledge-domains using the built-in domain-editor.

The fact, however, that questions are answered by generating and evaluating Prolog expressions, limits Masque's applicability to ad hoc Prolog databases. Existing commercial (usually relational) databases cannot be accessed, and thus the system cannot be used in most real-life applications. This paper describes an extended Masque version, called Masque/sql, which can be used as a front-end to any commercial database supporting the SQL query lan-

*The present paper appears in the "Proceedings of the Sixth International Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems", Edinburgh, 1993, proceedings published by Gordon and Breach Publishers Inc., Langhorne, PA, U.S.A.

guage. Masque/sql answers the user's questions by generating suitable SQL code, which is executed by the relational DBMS (Database Management System).

The full linguistic coverage of the original Masque is still available, and the new system can be configured as easily as the old version. The configuration procedure is simple and semi-automated: the built-in domain-editor helps the user describe the entity types of the world to which the database refers, using an is-a hierarchy. The user is then assisted to declare the words expected to appear in the English questions, and to define the meaning of each declared word in terms of a logic predicate. The is-a hierarchy is used to restrict the possible argument types of the logic predicates, as in many-sorted logic. Each predicate is finally linked to a database table/view or SQL `select` statement.

English questions are transformed internally into expressions of a Prolog-like, meaning representation language, called LQL (Logical Query Language). LQL, derived from DCW [2], is roughly a subset of Horn logic. It is formally defined in [3], and can be used to express the meaning of a large set of English questions. An algorithm to efficiently translate LQL expressions into SQL has been designed, and a Prolog implementation of this algorithm is integrated into Masque/sql. The algorithm itself is an example of automated translation from Prolog to SQL.

Tests made with a sample Ingres database containing world geography data, showed that Masque/sql can answer the user's questions almost as fast as the original Masque version. Masque/sql answered most test questions in under six seconds CPU time, including the time spent by the DBMS to retrieve data from Ingres.

Although the test database was small, the performance of Masque/sql should not be affected significantly when using larger databases: Masque/sql transforms each user question into a single SQL query, and the relational DBMS is left to find all answers to the query, utilising its own specialised optimisation and planning techniques. Thus, the full power of the relational DBMS is available during the question answering, and the system is expected to scale up easily to larger databases. In alternative approaches, the DBMS is only used to retrieve partial results, which are then joined by the natural language front-end, using some simplistic strategy. The efficiency of front-ends based on such approaches can deteriorate significantly as the size of the database increases.

Masque/sql seems to be efficient, friendly and portable enough, to be tested in real-life applications. Currently, only Ingres for UNIX is supported, but Masque/sql could be easily ported to any DBMS that supports C with embedded dynamic SQL.

Masque/sql is the outcome of a research project described in [3]. Information about the original version of Masque can be found in [4].

3 Entities and Relations

Masque/sql assumes that the world to which the English questions refer consists of entities and certain kinds of relations between entities. The possible entity types of the world are organised in a forest of is-a trees, where each ancestor subsumes its descendant types. In figure 1, `person`, `staff`, and `feature` are entity types. A `staff` entity is also of type `person`, and a `person` is either `customer` or `staff`.

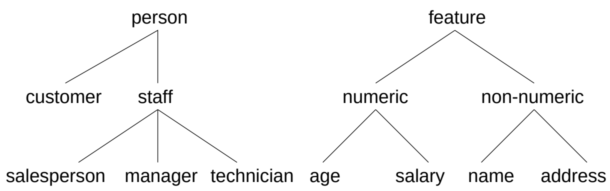

Figure 1: An is-a type-hierarchy

The meaning of each word can be expressed as a logic predicate, the arguments of which correspond to world entities. For example, the meaning of the noun *"manager"* (as in *"Is PG a manager?"*) is expressed as `is_manager(M)`, where `M` is a variable corresponding to an entity of type `manager`. Similarly, the meaning of the noun *"salary"*, as in *"What is the salary of each manager?"*, is expressed as `has_salary(Sal,Stf)`, where `Sal` is of type `salary`, and `Stf` is of type `staff`. The second argument of `has_salary` is declared to be `staff`, because we wish to be able to use the noun *"salary"* with any entity of type `salesperson`, `manager`, or `technician` (e.g. *"What is the salary of each technician?"*).

In MASQUE/SQL, the meaning of the words is expressed using two types of predicates:

Type-A predicates express relations between individual entities. For example, the type-A predicate `has_salary` links entities of type `staff` to entities of type `salary`.

Type-B predicates express relations linking entity-sets to entities. For example, the meaning of *"average"*, as in *"What is the average age of the managers?"*, is `av(Aver,Set)`, where `Set` stands for a set of `numeric` entities, and `Aver` stands for a `numeric` entity corresponding to the average of `Set`. In other words, `av` links sets of `numeric` entities to individual `numeric` entities (their average).

The is-a hierarchy rules out many inappropriate questions. For example, in *"What is the salary of each customer?"*, the meaning of *"salary"* is expressed as `has_salary(S,C)`, where `C` is of type `customer`. The second argument of `has_salary` is declared to be of type `staff`, and `customer` is not subsumed by `staff`. The system can, therefore, reason that the question is ill-formed. In principle, it could also report it does not know how to compute the salary of `customers`.

4 System Architecture

Figure 2 shows the MASQUE/SQL architecture. The English question is first parsed by an Extraposition Grammar parser [5]. The resulting parse tree is processed by the semantic interpreter, to become an expression in the LQL meaning representation language. The LQL expression is then translated into an SQL query, which is executed by the DBMS against the relational database. The response generator uses the results of the SQL query, to report the answer or a reasonable ignorance message.

The lexical dictionary lists all the possible forms of the words expected to appear in the user's questions. Common words like articles and auxiliary verbs are built-in and do not need to be declared. The meaning of each word listed in the lexical dictionary has to be described in the semantic dictionary, in terms of a logic predicate, as described in the previous section. MASQUE/SQL uses these logic predicates to form LQL queries, expressing the meaning of the corresponding English questions. For example, *"What is the salary of each manager?"* receives the LQL query:

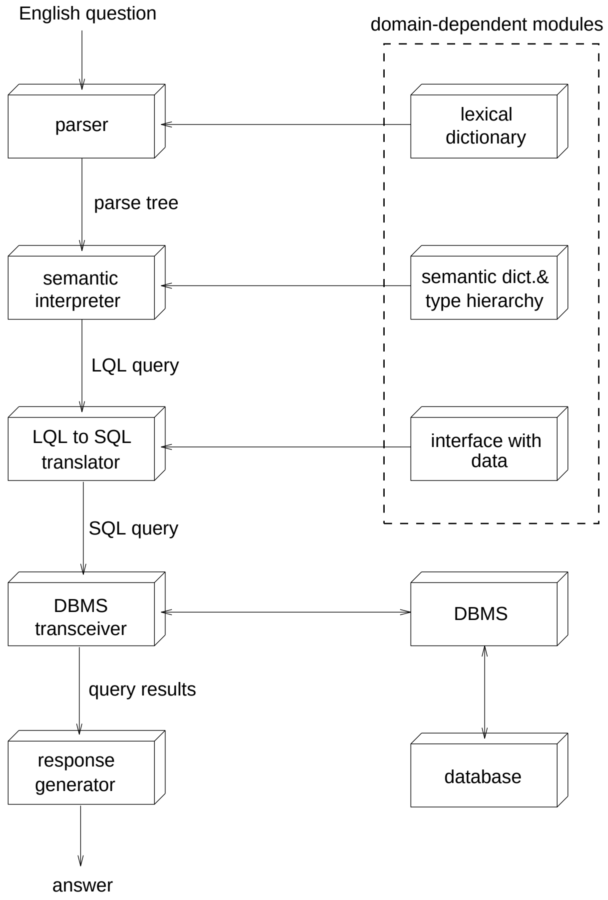

Figure 2: The Masque/SQL architecture

```
answer([S,M]):- is_manager(M),
                has_salary(S,M)
```

where `is_manager(M)` is derived from the noun *"manager"*, and `has_salary` from the noun *"salary"*.

LQL is a subset of Prolog. The LQL syntax is much simpler than the full Prolog syntax, and the semantics of LQL are weaker than Prolog's. For example, only variables, atoms, and certain forms of terms may appear as arguments of an LQL predicate, and the body of an LQL query need not be evaluated using Prolog's depth-first strategy.

Each type-A predicate of functor `func` and arity `n` is automatically assumed to correspond to a database table/view `func#n(arg1,arg2,...,argn)`. For example, the type-A predicate `has_salary` is assumed to correspond to the database table or view `has_salary#2(arg1,arg2)`, where `arg1` stands for the salary and `arg2` for the employee's id. If the tables in the database do not have the form required by Masque/sql, SQL views can be used to convert to the necessary forms.

Type-B predicates are linked, in the interface with data, to pseudo-SQL queries acting over the imaginary relation

```
pair_set(first, second)
```

For example, the type-B predicate `av` (average) is linked to the query:

```
SELECT avg(first) FROM pair_set
```

The meaning of these pseudo-SQL queries will become clearer in the following section.

The two dictionaries, the type hierarchy and the interface with data are the only modules that have to be reconfigured whenever the system is ported to a new knowledge-domain/database. Both the dictionaries and the type hierarchy are created automatically, by answering the questions of the built-in domain-editor. The domain-editor also provides information that assists the user to create the interface with data.

5 Translating to SQL

The simplicity of the LQL syntax and semantics allows the translation from LQL into SQL to be relatively simple. A complete description of the translation algorithm used in Masque/sql is out of the scope of this paper. Instead, the main ideas will be highlighted using an example. This section assumes that the reader is familiar with SQL.

"What is the average age of the managers?" receives the LQL query:

```
answer([Aver]):-
   setof(Age:Mng,
         (is_manager(Mng),
          has_age(Age, Mng)),
         Ages_list),
   av(Aver, Ages_list)
```

which denotes that the system must compute all pairs of the form

$$< age\_of\_manager, manager >$$

and report the average of the *age*s in the pairs.

The translation algorithm uses a bindings structure, which maps LQL variables to SQL code fragments. Whenever a new LQL variable is encountered, a suitable SQL fragment is stored as the binding of that variable. When the variable is reencountered, its binding is used to create a `WHERE` condition. Returning to our example, the type-A predicate `is_manager(Mng)` is translated into:

```
SELECT *
FROM is_manager#1 rel1
```

and `rel1.arg1` becomes the binding of `Mng`. Then `has_age(Age,Mng)` is processed. The binding of `Mng` (i.e. `rel1.arg1`) is used to create a `WHERE` condition:

```
SELECT *
FROM has_age#2 rel2
WHERE rel2.arg2 = rel1.arg1
```

and `rel2.arg1` becomes the binding of `Age`. (Intermediate results of the translation algorithm are not always well-formed SQL queries.)

To translate a conjunction list, the `FROM` and `WHERE` parts of the translations of the conjuncts are merged. In our example, the second argument of `setof` becomes:

```
SELECT *
FROM is_manager#1 rel1,
     has_age#2 rel2
WHERE rel2.arg2 = rel1.arg1
```

The processing of the overall `setof` instance causes the binding of `Ages_list` to become:

```
SELECT rel2.arg1, rel1.arg1
FROM is_manager#1 rel1,
     has_age#2 rel2
WHERE rel2.arg2 = rel1.arg1
```

The `SELECT` part above is generated by observing the first argument of `setof` (`Age:Mng`) and by using the bindings of `Age` (`rel2.arg1`) and `Mng` (`rel1.arg1`).

The type-B predicate `av` is linked to the pseudo-SQL query:

```
SELECT avg(first)
FROM pair_set
```

Consulting the `SELECT` part of the binding of `Ages_list`, `first` can be associated to `rel2.arg1`, and `second` to `rel1.arg1`. Processing the `av` instance causes the binding of `Aver` to become:

```
SELECT avg(rel2.arg1)
FROM is_manager#1 rel1,
     has_age#2 rel2
WHERE rel2.arg2 = rel1.arg1
```

where the `FROM` and `WHERE` parts are the same as in the binding of `Ages_list`. The translation of the full LQL query is the binding of `Aver`.

6 Assessment

MASQUE/SQL was tested with a sample world-geography Ingres database on a heavily loaded Sequent Symmetry machine. The database contained 10 tables, 5 rows the smallest and 856 rows the largest. The lexical dictionary contained 86 user-defined words.

The system was able to answer complex English questions, in almost all cases under six seconds (including the time spent by the DBMS). The system's efficiency is mainly due to the fact that questions are answered by generating a single SQL query, which is executed by the DBMS, utilising its advanced planning, indexing and optimising techniques.

One shortcoming of the current system is that the answers are less informative than the ones generated by the original MASQUE, in cases of failed queries. Consider the question *"What is the salary of each secretary in building1?"*, and let us assume that the database contains no data about employees in `building1`. The LQL query is:

```
answer([Sal, Secr]):-
   is_secretary(Secr),
   located(building1, Secr),
   has_salary(Sal, Secr)
```

In the original MASQUE, LQL queries are executed as Prolog programs. When the goal `located(building1, Secr)` fails, the system keeps an internal note of the failed goal, which helps the response generator produce a suitable ignorance message. In MASQUE/SQL the LQL query is transformed into an SQL query. If the SQL query fails, the system does not know which part of the query caused the failure, and thus it can only produce a less informative message. A possible solution would be to translate and execute subparts of the LQL query, until the subpart causing the failure is discovered.

In MASQUE/SQL each logic predicate is mapped directly to a database table, view, or query. This sometimes causes redundant joins to appear in the SQL queries. *"Which country's capital is London?"* receives the LQL query:

```
answer([C]):- capital_of(london, C),
              is_country(C)
```

Since the second argument of `capital_of` is always a country, the `is_country` instance is redundant. The `is_country` causes a redundant join in the SQL query:

```
SELECT rel1.arg1
FROM capital_of#1 rel1,
     [is_country#2 rel2]
WHERE rel1.arg1 = 'london'
      [AND rel1.arg2 = rel2.arg1]
```

Square brackets denote redundant code caused by the redundant `is_country` in the LQL query. The type hierarchy could be used to remove redundant logic predicates from the LQL queries [3].

7 Conclusions

Despite its limitations, MASQUE/SQL seems to be powerful, portable, and friendly enough to be tested in real-life applications. The fact that it can be used with any database supporting SQL, and its simple, semi-automated configuration procedure, make it a valuable tool for experimenting with natural language front-ends in practice.

References

[1] D. Warren and F. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *Computational Linguistics*, 8(3-4):110–122, 1982.

[2] F. Pereira. *Logic for Natural Language Analysis*. PhD thesis, Dept.of AI, Univ. of Edinburgh, 1982. Also as SRI Tech.Note 275, 1983.

[3] I. Androutsopoulos. *Interfacing a Natural Language Front-End to a Relational Database*. Tech. Paper no.11, Dept.of AI, Univ.of Edinburgh,1993 .

[4] P. Auxerre and R. Inder. *MASQUE Modular Answering System for Queries in English - User's Manual*. Tech.Rep. AIAI/SR/10, AI Applications Institute, Univ. of Edinburgh, 1986.

[5] F. Pereira. Extraposition Grammars. *Computational Linguistics*, 7(4), 1981.