

# Using SQL with Prolog to improve performance with large databases

Kevin Boone  
Department of Computing Science  
Middlesex University, London

February 1999

## Abstract

This article describes how the performance of certain Prolog programs can be improved by storing large lists of facts in an SQL database rather than as Prolog facts. In experiments that will be described, the speed improvements ranged from negligible to a factor of over 200. This improvement comes about because SQL servers are strongly optimized for searching large, flat tables. However, modern Prolog compilers are increasingly good at handling large volumes of data, and not all applications will benefit from the use of SQL. This article suggests which types of application are likely to benefit, and discusses other ways in which the use of SQL may be of benefit in Prolog programming.

## 1 Introduction

### 1.1 Prolog

A Prolog program can be viewed as the specification of a list of goals to be satisfied. For example, the goal

```
?- dog(X).
```

can be interpreted as ‘find something that’s a dog and call it X’. This goal can be satisfied if there is an entry in the program’s database of the form `dog(rovers)`.

which is a *fact*. The goal may perhaps be satisfied by a *rule* like

```
dog(X) :- mammal(X), hastail(X), hasfur(X), quadruped(X).
```

if there are facts or further rules concerning mammals, tails, fur, and number of legs in the database. This rule can be interpreted broadly as ‘X is a dog if X is mammal, a quadruped and has fur and a tail’.

Some Prolog programs depend heavily on facts. This is particularly true in natural language processing applications. For example, a system to detect proper nouns in text [Wakao et al., 1996] used a fact database with about 5,500 static facts (names of organizations, people, titles, associations etc).

Elementary natural language parsers of the type described by [Clocksin and Mellish, 1987] generally resolve sentences into trees whose ‘leaves’ are of facts of the form

```
noun(aardvark).  
noun(zebra).  
verb(walks).  
/* etc */.
```

Clearly, for such a system to have a substantial vocabulary, it must have access to a many such facts. To determine whether a particular fact is asserted it is necessary for Prolog to search the database, perhaps linearly from end to end. Execution of a program may require many thousands of such searches.

While Prolog can of course determine whether a goal such as `noun(X)` matches the fact `noun(orange)`, this is essentially a database lookup job, and there are systems which are highly optimized for carrying out such lookups. It seems possible therefore, that integrating such a system into a Prolog program may lead to significant performance gains.

## 1.2 SQL

Structured query language (SQL) is the standard programming interface to database systems that are arranged as collections of tables. An SQL database server accepts SQL queries, and returns sets of matching database entries. A lookup query has the form

```
select partofspeech from lexicon where word='orange'
```

meaning ‘get the contents of the “partofspeech” column from all rows of the table “lexicon” where the “word” column contains “orange”’. SQL servers perform only this highly specialized task, and modern versions are extremely good at it. In addition, an SQL server does not expect the complete database to be loaded into the computer’s memory (Prolog generally does), allowing databases with millions of entries to be handled. A further useful feature is that SQL is designed for client-server operation, where a number of applications or computers share the same centralized database.

## 1.3 Comparing Prolog to SQL

Prolog is *deductive*, that is, it can reason using facts and rules. SQL databases are simply a collection of data tables. In both languages database search is a central feature, but Prolog searches can be far more complex, as they may involve variables. Nevertheless many Prolog applications involve straightforward searches matching items in a database, which is an area where SQL excels.

## 1.4 Extending Prolog with SQL

Given the discussion above, it seems reasonable to hope that significant performance advantages may be achieved for a Prolog program by allowing straightforward ‘fact lookups’ to be replaced with SQL queries. This article describes a simple investigation whether such advantages are realized in practice and, if so, how significant they are likely to be. It should be noted that systems based on a combination of SQL and Prolog are already in use. For example, [Arner, 1994] describes such a system for management of large volumes of information in the newspaper industry. However, in Arner’s application the roles of Prolog and SQL are largely distinct (data validation and data management), and the method of integration is not described in detail.

## 2 Methods

### 2.1 Environment

The experiments to be describe were all carried out in the following environment.

- SWI-Prolog version 3.1.2 [Wielemaker, 1998], a Prolog compiler that is available free of charge for academic use.
- MySQL version 3.2.1 [Widenius, 1998], an SQL server and client library that is available free of charge for academic use.
- GNU C++ compiler version 2.7.2.2
- A PC with single 233 MHz Pentium CPU running Linux version 2.0.35

### 2.2 Implementation

Like most modern Prolog compilers, SWI-Prolog provides facilities for defining new predicates in native machine code, as well as in terms of other Prolog predicates. MySQL provides a software library that can be linked with a C or C++ program to allow it to execute SQL queries. With these facilities, it was straightforward to implement three additional predicates that provide SQL database access. In the description below I have followed the standard Prolog convention of labelling arguments that are inputs to a predicate with a ‘+’ and outputs with a ‘-’.

- `connect_database(+Server, +UserID, +Password, +DatabaseName, -Handle)`. Makes a connection to the specified database on the specified server using UserID and Password as credentials. ‘Handle’ is unified with an integer that identifies the connection for use in other predicates.

- `query_database(+Handle, +Query, -Result)`. Executes the specified SQL query on the previously-opened database specified by 'Handle'. The results are returned in the form of a list of lists, where each sub-list represents a row from the selected table. Not all queries return results (e.g., 'update' queries), and these are represented by an empty list.
- `disconnect_database(+Handle)`. Disconnects from the SQL server and frees any allocated resources.

Implementation of these predicates required only 300 lines of C++; the result is a dynamically linkable library (called 'plsql') which can be loaded into the prolog compiler using the predicate `load_foreign_library(foreign(plsql))`. The library, with source code and documentation, can be obtained from <http://www.cs.mdx.ac.uk/staffpages/kevin/plsql-index.html>

### 2.3 Experimental queries

To test the efficacy of making fact lookups using SQL queries in Prolog, I used a database of 109,000 English words. In one set of experiments these were represented as rows of an SQL table, and retrieved by SQL queries using the predicates described above. In the other they were represented as prolog facts of the form

```
word(aardvark).
/*...*/
word(zebra).
```

With this arrangement, the complete fact list was compiled before any queries were executed. The compilation itself took approximately 11 seconds, but of course this only has to be done once per session.

The first query took as input a list of 600 words, picked randomly but with a uniform distribution of initial letter. This was to ensure that the complete database was exercised. The query extracted from the input all the words that did not appear in the database (like a spelling checker). The second query used a list of 600 *identical* words (actually the word 'zebra', this being towards the end of the database). This tests how well the Prolog compiler can optimize by caching queries. The third query constructed a Prolog list of all words that start with 'a' and those that start with 'z'. The difference between this and the first two queries lies in the way the database entries are processed. In the first two queries we are doing simple word lookup; the word either appears in the database or it does not. In the second we are matching words against database entries. In a sense this is a more difficult job, as the whole database needs to be enumerated and each item checked.

Query	With SQL lookup	With Prolog facts
Random word lookup	0.15 sec	39 sec
Fixed word lookup	0.10 sec	0.15 sec
Extract words starting with 'a' and 'z'	0.8	2.2

*Table 1: times taken to execute different types of query*

### 3 Results

The times taken to perform the three queries described above, with an SQL database and with standard Prolog facts, are summarized in table 1.

Note that times of 0.1 second are close to the timing granularity of the system, and one should be careful not to over-interpret differences in times smaller than this.

### 4 Discussion

The most obvious difference between the SQL-based query and the standard fact-based query corresponds to looking up random words in the database. Without SQL, it took 39 seconds to look up 600 words, compared to 0.15 seconds with SQL. Even 39 seconds is a creditable performance – about 15 words per second – with a flat database of over 100,000 words, but it can be seen that the SQL-supported version is much faster. However, the difference in speed is much less obvious when the same word is lookup up repeatedly. Presumably the prolog compiler is able to optimize this query by caching database lookups. In an application that handles ordinary text, where some words are repeated and some are not, the speed improvement will probably lie somewhere between the two extremes investigated here.

When extracting words that match a pattern, the SQL version is about three times as fast as the non-SQL version of the query.

None of the queries investigated here really *do* anything with the data retrieved. In an application like parts-of-speech tagging, where simple database lookup is likely to be a primary part of the the process, the improvements offered by SQL support may be fully realized. However, where significant logical processing is required, database lookup is less likely to be the limiting factor in performance. Of course, there may be other reasons for using an SQL database with Prolog apart from performance, e.g., to allow database sharing.

SWI-Prolog is not claimed to offer the ultimate performance possible from a Prolog compiler; on the contrary its main design goals are portability and ease of development. It offers some facilities to improve the speed of searches, such as indexing of facts, but it is possible that other compilers may offer speeds closer to that of SQL-based searches.

Nevertheless, this investigation shows that the use of SQL queries in certain Prolog programs may lead to a useful performance increase, and it is not at all difficult to provide the necessary facilities.

## 5 Acknowledgements

Thanks to Mike Wing and John Platts in the School of Computing Science for their helpful comments on the draft of this document.

## References

- HF Arner. Prolog for predicting advertisement costs in newspapers. In *Proceedings of the Practical Applications of Prolog conference, April 1994*, London, 1994. Available at [www.logic-programming.org/Arner\\_Howard/pap94.htm](http://www.logic-programming.org/Arner_Howard/pap94.htm).
- WF Clocksin and CS Mellish. *Programming in Prolog*. Springer-Verlag, New York, third edition, 1987.
- T Wakao, R Gaizauskas, and Y Wilks. Evaluation of an algorithm for the recognition and classification of proper names. In *Proceedings of the 4th international conference on computational linguistics*, pages 418–423, Copenhagen, 1996.
- M Widenius. *MySQL reference manual*. TCX DataKonsulter AB, 1998. Available at [http://www.tcx.se/mysql\\_docs/toc.html](http://www.tcx.se/mysql_docs/toc.html).
- J Wielemaker. *SWI-Prolog reference manual*. University of Amsterdam, Department of Social Science Informatics, 1998. Available at <ftp://swi.psy.uva.nl/pub/SWI-Prolog/>.