A logical approach to working with Supermarket databases

Aynoor Saleem
Computer Science Department, Stony Brook University, Stony Brook, NY

December 21, 2017

Abstract

Prolog is well suited for code development in areas that require working with, managing and analyzing high volume of data. In this project I present *supermarket_db*, which is a system implemented in Prolog that gives easy access to high quality data in the format of Prolog fact files. I have extended the paper to not only work with biological datasets but another one too i.e. Supermarket datasets. Moreover, I have evaluated their work and verified their claims for both cases of *bio_db* and how they apply to *supermarket_db*.

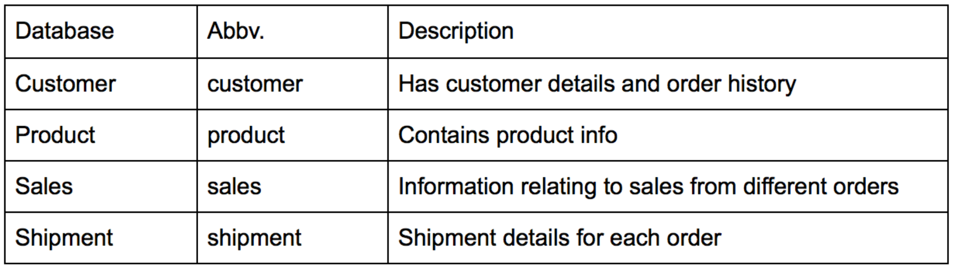
1 Introduction

This project report describes the design structure and potential of an extensible system for managing and working with supermarket databases or any other. The included features are given the datasets the user can easily load the data into Prolog fact files and have them loaded in the program for user to further query upon. The datasets included should be in the right *.csv* format for the program to read them correctly. The running of the program is very intuitive so that the user can get relevant information from high volume of data using simple prologue predicates.

Code can be found here.

2 Datastets

The purpose of this project is to extend the concept of *bio_db* to be used in other areas that also require managing and working with high volume of data. For this work a databases related to supermarket information have been chosen. Figure 1 shows a summary of databases supported.



Database	Abbv.	Description
Customer	customer	Has customer details and order history
Product	product	Contains product info
Sales	sales	Information relating to sales from different orders
Shipment	shipment	Shipment details for each order

Figure 1: Supported supermarket databases.

For all these databases they map information between different details of the order placed in the supermarket. Prolog facts of arity 2 are used to show this relationship between different columns of the database. In the case where bi-directional translation is required fact database that reverse the order of arguments is constructed.

- Customer database includes information about the customer like the name, customer id and the segment where the customer belongs to e.g. consumer or corporate. It also has the order ids if the orders made by each customer.
- Product database has for each order what was its product id, product name, the category and subcategory the product belonged too.
- Sales database includes for each order the total sales, discounts, quantity and profits.
- Shipment database has complete information of when the order was shipped and where was it destined for.

For this kind of data it is very useful to have a simple interface that allows quick and easy analysis. Through Prolog this is achieved.

3 Data management

In supermarket_db the native representation of the order knowledge described above is as Prolog facts. The library presents those facts to the programmer as a unifying level of abstraction. Beneath this, there is a mechanism via which the data are delivered to the predicates i.e. Prolog fact files.

3.1 Predicate naming

An example of a map predicate is

```
map_customer_custid_custname( CustId, CustName ).
```

The predicate translates between customer identifiers and customer names. The predicate name consists of 4 components, the first of which determines the type of data, which in this case is a map. The second component, customer corresponds to the source database and the third component, custid, identifies the first argument of the map to be the unique identifier field for each customer (here a positive integer starting at 1 and with no gaps). The last part of the predicate name corresponds to the second argument, which here is the name of the customer. The following interaction shows how the predicate can be used to find the symbol of a gene given its HGNC identifier.

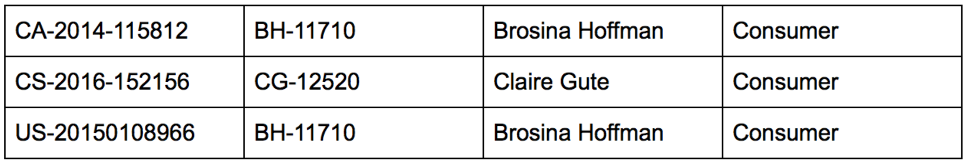
```
?- map_customer_custid_custname('DV-13045', Name).
     Name = 'Darrin Van Huff'.
```

3.2 Data Serving methods

The main mechanisms via which library's data predicates can be stored and served is Prolog fact files. This requires in-memory loading for serving, thus it requires more memory and time for loading. Prolog fact files are used when big time consuming searches are required as the Prolog facts are extremely fast particularly when requests for data instantiate the first argument of their call. One additional considerations is that the fact that the Prolog facts are stored in plain text files which can be helpful when debugging.

All data predicates loaded once the program is loaded as such. The database files are read and prolog fact files are made. These are then loaded into the program. The entire process is transparent to the user.

```
?- [supermarket\_db].
true.
?- map_product_orderid_productname(Oid, ProductName).
Oid = 'CA-2016-152156',
ProductName = 'Bush Somerset Collection Bookcase' ;
Oid = 'CA-2016-152156',
ProductName = 'Hon Deluxe Fabric Upholstered Stacking Chairs, Rounded Back' ;
Oid = 'CA-2016-138688',
```



CA-2014-115812	BH-11710	Brosina Hoffman	Consumer
CS-2016-152156	CG-12520	Claire Gute	Consumer
US-20150108966	BH-11710	Brosina Hoffman	Consumer

Figure 2: Gene ontology terms and associated GO term names for LMTK3. Third column shows the total of genes in the GO term

```
    ProductName = 'Self-Adhesive Address Labels for Typewriters by Universal' ;
    Oid = 'US-2015-108966',
    ProductName = 'Bretford CR4500 Series Slim Rectangular Table' ;
    Oid = 'CA-2014-115812',
    ProductName = 'Eldon Expressions Wood and Plastic Desk Accessories, Cherry Wood' ;
    Oid = 'CA-2014-115812',
    ProductName = 'Newell 322'...
```

4 Example

Customers belong to different segments like corporate or consumer etc. Here for all the orders we will fond the customer details i.e the name, id and which segment does the customer belong too.

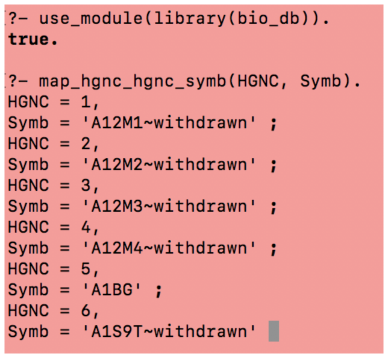
```
 write_list([]).
 write_list([H|T]):-
        writeln(H),
        fail;
        write_list(T),
        true.

find_customer_id_name_segment([], []).
find_customer_id_name_segment([Oid|T], [(Oid,Cid, Cname, Segment)|L]):-
        map_customer_custid_orderid(Cid, Oid),
        map_customer_custid_custname(Cid, Cname),
        map_customer_orderid_segment(Oid, Segment),
        find_customer_id_name_segment(T, L).

order_customer_info():-
        findall(Oid, map_product_orderid_category(Oid, 'Furniture'), Oids ),
        sort(Oids, UOids),
        find_customer_id_name_segment(UOids, L),
        write_list(L).
```

5 Evaluation

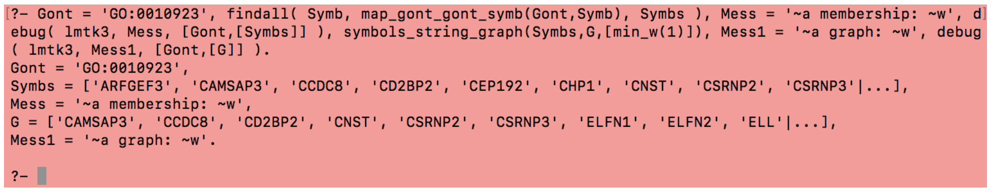
I ran most of their examples which seem to be working fine. The results of running some are shown below:
From the databases I verified the results.



```
?- use_module(library(bio_db)).
true.

?- map_hgnc_hgnc_symb(HGNC, Symb).
HGNC = 1,
Symb = 'A12M1~withdrawn' ;
HGNC = 2,
Symb = 'A12M2~withdrawn' ;
HGNC = 3,
Symb = 'A12M3~withdrawn' ;
HGNC = 4,
Symb = 'A12M4~withdrawn' ;
HGNC = 5,
Symb = 'A1BG' ;
HGNC = 6,
Symb = 'A1S9T~withdrawn'
```

Figure 3: Find symbols for each unique hgnc identifier



```
?- Gont = 'GO:0010923', findall( Symb, map_gont_gont_symb(Gont,Symb), Symbs ), Mess = '~a membership: ~w', d
ebug( lmtk3, Mess, [Gont,[Symbs]] ), symbols_string_graph(Symbs,G,[min_w(1)]), Mess1 = '~a graph: ~w', debug
( lmtk3, Mess1, [Gont,[G]] ).
Gont = 'GO:0010923',
Symbs = ['ARFGEF3', 'CAMSAP3', 'CCDC8', 'CD2BP2', 'CEP192', 'CHP1', 'CNST', 'CSRNP2', 'CSRNP3'|...],
Mess = '~a membership: ~w',
G = ['CAMSAP3', 'CCDC8', 'CD2BP2', 'CNST', 'CSRNP2', 'CSRNP3', 'ELFN1', 'ELFN2', 'ELL'|...],
Mess1 = '~a graph: ~w'.

?-
```

Figure 4: For the gene ontology term ”GO:0010923’ find all its symbols