4.900

ProData Interface

by Rob Lucas

Prodata Interface

The contents of this manual describe the licensed products, ***WIN-PROLOG*** Prodata Interface, and are believed correct at the time of going to press. They do not embody a commitment or imply any liability on the part of Keylink Computers or Logic Programming Associates (LPA), who may from time to time make changes to the specification of the products, in line with their policy of continual improvement. No part of this manual may be stored, reproduced or transmitted in any form, electronic or mechanical, for any purpose other than the licensee's personal use, without the prior written agreement of Keylink Computers.

19 July, 2004

CONTENTS

Logic Programming Associates Ltd.
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London SW18 3SX
England

phone: +44 (0) 20 8871 2016

fax: +44 (0) 20 8874 0449

web site: http://www.lpa.co.uk

CONTENTS

CONTENTS....4
Chapter 1 - INTRODUCTION....7
Chapter 2 - INSTALLATION....9
Installing Prodata....9
What Does Prodata Consist Of?....9
Setting Up The Data Sources....9
Loading Prodata....9
Examples....10
Chapter 3 - USING THE INTERFACE....11
Connecting to a DBMS....11
Disconnecting from a Database....12
Ending a Prolog session....13
Accessing a database table....13
Augmenting queries for attached tables....14
Specifying connection for db_attach....15
Quoting An SQL Query....15
Type Conversions....17
From Prolog to DBMS....17
From DBMS to Prolog....18
Joins....19
Relational Operators....20
Programmatically Generating An op/1 Term....21
Reading An op/1 Term....21
Alternative access to DBMS tables....21
Building An SQL Query Up As a List....23
Cursor based access....24
Long fields....25

CONTENTS

Chapter 4 - DATA DICTIONARY....26
Data sources....26
Database tables....26
Table arity....28
Table columns....28
Table tuples....30
Data dictionary tables....30
Chapter 5 - OTHER DATABASE OPERATIONS....31
Creating tables and indexes....31
Deleting tables, views and indexes....32
Adding and deleting records....32
Handling Prolog Lists....33
Other SQL commands....34
Seeing the SQL for a generated query....34
Seeing DBMS error messages....35
Setting default connection....35
Matching constants with nulls....36
Chapter 6 - TRANSACTION MANAGEMENT....37
Transactions....37
Chapter 7 – SQL EXAMPLES....39
Chapter 8 – PRODATA AND STANDALONE APPLICATIONS....45
Chapter 9 – CONNECTING TO EXCEL FILES....46
Chapter 10 - TROUBLE SHOOTING....49
Prolog Error....49
Database Interface Error....50
The ODBC Administrator's Tracing Facility....50
The Debugging Version of LPADBW.DLL....50
TRACE.PL....50
ODBC Drivers....51
Chapter 11 - SUMMARY OF PREDICATES....52

CONTENTS

Introduction 52
The predicates 52
INDEX 66

Chapter 1 - INTRODUCTION

This document describes the Prodata LPA Prolog interface which is a tight coupling between ***LPA-PROLOG*** for Windows and all DataBase Management Systems (DBMSs) which support a sufficient level of Open DataBase Connectivity (ODBC) compliance to be used with Microsoft ODBC 2.

Prodata allows database tables to be accessed from Prolog as though they existed within Prolog's environment as unit ground clauses (facts). This facilitates the use of Prolog rules over the contents of the database, with no need to download any part of the database, as all database accesses are done 'on-the-fly'.

Backtracking, cut, call, not and all other standard Prolog mechanisms work identically over the table accesses and the internal database, thus achieving the highest level of transparency possible.

This software architecture allows a completely different style of database programming that leaves fourth generation languages way behind. All the benefits inherent in Prolog development can now be transferred to the database field without any disadvantages.

In particular Prodata offers the following functionality:

Concurrent access and update, with program control;

Full data access transparency;

All database functions can be carried out from within Prolog, such as creating tables, adding/deleting records, allocating access rights; in fact whatever may be achieved via normal Structured Query Language (SQL), just as long as the ODBC driver supports it;

Optional automatic display of error messages produced by the database system;

Automatic data conversions;

Full access to the data dictionary, for those DBMSs which have one;

Completely transparent cursor manipulation: there is never any need for your program to manipulate the table cursors.

It is possible to see the SQL generated for a Prolog access to the DBMS by the use of specially provided predicates.

Date types are handled uniformally across all Relational DataBase Management Systems (RDBMS).

All DBMSs with ODBC compliant drivers are now supported.

Simultaneous connection to multiple DBMS is now possible, allowing table joins across database systems.

The interface is described from the user (Prolog programmer) perspective in the following chapters. All facilities are available for all the supported DBMSs unless otherwise stated; the supported DBMSs include:

Access

SQL Server

Oracle

DB2

dBase

Excel

Chapter 2 - INSTALLATION

Installing Prodata

To install the Prodata toolkit correctly, you should run the setup program, SETUP.EXE, on the *WIN-PROLOG* CD-ROM. For more details on the installation process please refer to the "*WIN-PROLOG* User Guide".

What Does Prodata Consist Of?

The Prodata interface consists of two files: **DBLINK.PC** (in the SYSTEM directory), which contains the declarations and Prolog object code for the main part (front-end) of the interface, and a DLL: **LPADBW.DLL** (in the *WIN-PROLOG* root directory) , which contains executable code which allows access to the various DBMSs via ODBC drivers (i.e. it is used by the Prolog object code to communicate with the ODBC DLLs).

If you create a standalone application in a directory other than *WIN-PROLOG*'s root directory, you will need to make sure that **DBLINK.PC** and **LPADBW.DLL** are accessible.

Setting Up The Data Sources

For each data source you wish to access through Prodata you will need to have configured it as a Data Source using the ODBC Data Source Administrator. The ODBC Data Source Administrator is reachable via the Windows Control Panel. Before you can configure a data source in the ODBC Data Source Administrator, you need to have the appropriate ODBC driver installed for the data source. Consult the Windows documentation for how to do this.

If you are using Prodata within a ProWeb or WebFlex application, the data source must be set up as a System DSN.

Loading Prodata

The Prodata Prolog code can be loaded into *WIN-PROLOG*, and the DLL launched, by executing the following at the *WIN-PROLOG* command line:

```
?- ensure_loaded( system(dblink) ).<return>
yes
```

You can now access your external database sources using the predicates described in the following chapters.

If you are accessing a PC-based database, such as dBASE or Paradox, then you may need to have the DOS program **share.exe** loaded to perform some operations.

Examples

Prodata is supplied with a few examples in the EXAMPLES\PRODATA directory:

NWIND.PL - This requires the NORTHWIND data source which comes with Microsoft Access and Microsoft SQL Server.

DEPT.DBF

EMP.DBF

Chapter 3 - USING THE INTERFACE

Connecting to a DBMS

In order to access a server-based DBMS, such as Oracle or Ingres, the appropriate communications driver (e.g. Oracle's SQL*Net) must be started prior to running Windows. You must also have been granted access to the DBMS by the System Manager or Database Administrator who will have given you, where appropriate: Login name, Password, Database name, Server name, or Connect string. If you can use other SQL tools on the PC to access the server, then you should be able to connect to the server from ***WIN-PROLOG***. For any particular database, you must familiarise yourself with the particular requirements of the ODBC driver that you are intending to use.

Connection to all DBMSs is established via the ***db_connect/1*** predicate. Its first argument is the Data Source Name (DSN) which identifies the data source as specified by the ODBC Windows Administration tool.

Two example dBASE files are supplied with Prodata which may be used to familiarise yourself with Prodata. To use them you must have a dBASE ODBC driver. Run the ODBC Administrator program from the Windows Control Panel. Select Add and select a dBASE driver. Fill in the Data Source Name as 'mydbf' and give the directory where you have installed the dBASE files for the database directory entry. Finally click on OK. You now have a Data Source called 'mydbf'.

Whenever you establish a database connection, any data dictionary information that is stored locally within Prolog for that database system is removed. This may be there from a previous session that was saved. The second argument returns the Host Database Connection (HDBC) number which is used for all subsequent database accesses relevant to this connection. For example, if I have a data source called 'mydbf' then I can connect to it with:

```
?- db_connect( mydbf, HDBC ).<return>
HDBC = 1
```

If the connection is successful the goal will succeed and HDBC will be instantiated to the connection number. It is possible to omit the second argument as in:

```
?- db_connect( mydbf ).<return>
yes
```

where you intend to use the default connection number in future operations. Virtually all Prodata predicates can be used with or without a specified HDBC. Where the HDBC is not specified the default value is used. Both versions of ***db_connect*** have the affect

of making the HDBC for the connection the current default. The default may be changed by using ***db_flag***, which is documented later.

You may augment the Data Source Name with 'attribute=value' pairs to tailor your connection. These attributes and values will be specific to the ODBC database driver that you are using. For example: using the Q+E dBASE driver, I can specify the character set and connect with:

```
?- db_connect( 'mydbf;Charset=ANSI' ).<return>
yes
```

For example, the data source (i.e. "mydb") you are trying to connect to may have a password (i.e. "rhubarb") set:

```
?- db_connect( 'mydb;PWD=rhubarb' ).<return>
yes
```

If your database allows spaces within column names then column names may need special treatment such as quoting. For example Access requires column names that have spaces to be enclosed within double quotation marks. This is effected by connecting in the following form:

```
?- db_connect( access(nwind) ).<return>
yes
```

The principal functor of the connection term indicates the database type being connected to (you may place any name here whatsoever). An additional entry must then exist in the database that tells the interface how column names are to be treated. In the case of Access the entry is:

```
?- db_col_spaces( access, `"`, `"` ).<return>
yes
```

This indicates that columns containing spaces must be double quoted for Access. This is actually already provided within the ProData interface for Access users. Similar facts can be asserted for other databases that require special treatment.

Disconnecting from a Database

You may disconnect from the default connection (HDBC) with:

```
?- db_disconnect.<return>
yes
```

or from a particular connection with:

```
?- db_disconnect( 1 ).<return>
yes
```

where the argument is the HDBC number. In either case the default connection is set to none and needs to be set either by a ***db_connect*** or ***db_flag*** goal before any predicate that relies on a default HDBC is used.

Ending a Prolog session

If you finish a Prolog session (using *halt/0* or selecting Close from the system menu) with one or more database connections established through ***db_connect*** still open, the Prodata interface will automatically close those connections.

Accessing a database table

The simplest way to access a table is to attach a Prolog predicate name to it and then treat it as though it were a set of Prolog facts. For example:

```
?- db_attach( dept, 'DEPT' ).<return>
yes
```

attaches to the table 'DEPT' for the default connection. This generates a rule, *dept/3*, which gives you access to the table. For example:

```
?- dept( Deptno, Dname, Loc ).<return>
Deptno = 10
Dname = 'ACCOUNTING'
Loc = 'NEW YORK'
```

Backtracking can be forced as usual by typing a semi-colon or pressing the <spacebar>.

The **db_attach** above causes a rule to be generated which has the head:

```
dept( P3, P2, P1 ):-
```

It can be seen by using the *listing/[0,1]* predicate. At the point of making the ***db_attach***, data dictionary information for the table is asserted into the local Prolog database to speed subsequent queries and in order to generate the necessary SQL for the table access. These are of the form:

```
?- data_dictionary( HDBC, Tname, Fieldno, Colname,
Coltype ).<return>
HDBC = 1 ,
Tname = 'Shippers' ,
Fieldno = 3 ,
Colname = 'Phone' ,
Coltype = 'CHAR' ;

HDBC = 1 ,
Tname = 'Shippers' ,
```

```
Fieldno = 2 ,
Colname = 'CompanyName' ,
Coltype = 'CHAR' ;

HDBC = Fieldno = 1 ,
Tname = 'Shippers' ,
Colname = 'ShipperID' ,
Coltype = 'NUMBER'
```

These can also be seen with the *listing/[0,1]* predicate.

Records with particular field values may be selected in the same way as Prolog clauses, i.e. the value is included in the term. For example:

```
?- dept( A, 'RESEARCH', C ).<return>
```

generates the query:

```
SELECT DEPTNO, LOC FROM DEPT WHERE DNAME='RESEARCH'
```

We can attach to a table called 'Order Details' with the following:

```
?- db_attach( order_details, '"Order Details"' ).<return>

?- order_details( A, B, C, D, E ).<return>
SQL> SELECT OrderID,ProductID,UnitPrice,Quantity,Discount
FROM "Order Details"
A = 10248 ,
B = 11 ,
C = 14 ,
D = 12 ,
E = 0 ;
```

If a field includes a quote then this should be represented by two quotes when using it as a selection value.

Augmenting queries for attached tables

It is possible to specify a string that is to be placed after the generated SQL which may be specific to the particular ODBC driver and/or table. This might be used by dBASE for specifying an index to use for improving performance. This augmentation string is given as a third argument to ***db_attach/3*** (or fourth argument to ***db_attach/4***) and must be an LPA string. For example, for the Q+E dBASE ODBC driver:

```
db_attach( dept, 'DEPT', `(dept.ndx/USE)` ).
```

This string will then be used for all table searches using dept.dbf via the attached predicate name '*dept/3*'. For example, the query:

```
dept(10,B,C).
```

now generates the SQL query:

select deptname,loc from dept where deptno=7782 (dept.ndx/USE)

which will then be interpreted by the Q+E dBASE ODBC. You must consult the documentation for the relevant ODBC driver in order to determine what strings can legally augment an SQL select query.

Note that a table may be attached to more than one Prolog predicate, and each '***db_attach***' may use different augmentation strings.

Specifying connection for db_attach

The connection number, HDBC, as returned by ***db_connect/2*** may be specified for ***db_attach*** in the form:

db_attach(HDBC, Pname, Table, Aug).

Note that when specifying the HDBC, you must also specify the Augmentation string, though this may be given as empty (``).

```
?- db_attach( 1, shippers, 'Shippers', `` ).<return>
yes
```

Using ***db_attach/3*** assumes the form given in the previous section, beware!

Quoting An SQL Query

There are three types of quotes that can be used in an SQL query:

Single quotes (i.e. '…')

Double quotes (i.e. "…")

LPA String quotes (i.e. `…`)

You can specify the entire SQL query as an atom or as an LPA string:

```
?- db_sql_select( 'SELECT * FROM CUSTOMERS', X ).
<return>

?- db_sql_select( `SELECT * FROM CUSTOMERS`, X ).
<return>
```

Within the SQL query, some items may need to be themselves quoted, such as when having a space character in a table name; the following table should help:

	Item within quoted as an atom (i.e. in '…' quotes)	Item within quoted as an LPA String (i.e. in `…` quotes)	Item within quoted in double (i.e. "…") quotes
Entire SQL query as an ATOM (i.e. in '…' quotes)	'Select * from "table" where "column" = ' 'value' ' ' i.e. 'Select * From "Customers" Where "Country" = ' 'Germany' ' '	'Select * from `table` where `column` = ' 'value' ' ' i.e. 'Select * From `Customers` Where `Country` = ' 'Germany' ' '	'Select * from "table" where "column" = ' ' value' ' ' i.e. 'Select * From "Customers" Where "Country" = ' 'Germany' ' '
Entire SQL query as an LPA STRING (i.e. in `…` quotes)	`Select * from "table" where "column" = 'value'` i.e. `Select * From "Customers" Where "Country" = 'Germany'`	`Select * from ``table`` where ``column`` = 'value'`  i.e. `Select * From ``Customers`` Where ``Country`` = 'Germany'`	`Select * from "table" where "column" = 'value'` i.e. `Select * From "Customers" Where "Country" = 'Germany'`
Entire SQL query in double (i.e. "…") quotes	NOT ALLOWED	NOT ALLOWED	NOT ALLOWED

NOTE: In the above table, for the sake of clarity, a space character has been placed between a pair of single quote characters (i.e. ' ' instead of ') to avoid confusion with a double quote character.

```
?- db_sql_select(`SELECT * FROM ``Employees`` WHERE
(HireDate={ts'1992-05-01 00:00:00'})`, X ). <enter>
SQL> SELECT * FROM `Employees` WHERE (HireDate={ts'1992-
```

```
05-01 00:00:00'})
X = [1,'Davolio','Nancy','Sales
Representative','Ms.',date([1948,12,8],[0,0,0]),date([199
2,5,1],[0,0,0]),'507 - 20th Ave. E.~M~JApt.
2A','Seattle','WA',98122,'USA','(206) 555-
9857',5467,'$address$'('0238C864:612'),'Education
includes a BA in psychology from Colorado State
University in 1970.  She also completed "The Art of the
Cold Call."  Nancy is a member of Toastmasters
International.',2]
```

Type Conversions

From Prolog to DBMS

When accessing a table, any value sent from Prolog to the interface is converted to an equivalent character string and used in constructing the necessary query, which is passed to the relevant ODBC Dynamic Link Library as a string. For most datatypes this is a very simple process. However some types are handled in specific ways. In particular, Datetime, date and time values use a specific Prolog structure in order to provide a standard mechanism which is the same for all databases.

In Prolog a Prodata datetime value is repesented by the structure:

date([Y,M,D],[H,Min,S,Dec])

The first argument of the date structure is the date, and the second argument is the time. The date is a list of three values, the year, the month, and the day. These should all be given as integers. The time is a list of four values, the hours, minutes, seconds and seconds decimal part, where the decimal part is optional. For a date only value, the time can be given as the empty list. For a time only value, the date can be given as an empty list. Examples are:

```
date([1991,12,1],[9,43,12,939492])

date([1812,7,30],[]).
```

When used for selection from a table, these are converted by Prodata to an ODBC standard, date, time or datestamp string.

```
?- db_tuple(employees,
[A,B,C,D,E,date([1948,12,8],[0,0,0]),F,G,H,I,J,K,L,M,N,O,
P] ). <enter>
SQL> SELECT
EmployeeID,LastName,FirstName,Title,TitleOfCourtesy,HireD
ate,Address,City,Region,PostalCode,Country,HomePhone,Exte
nsion,Photo,Notes,ReportsTo FROM employees WHERE
(BirthDate={ts'1948-12-08 00:00:00'})
A = 1 ,
```

```
B = 'Davolio' ,
C = 'Nancy' ,
D = 'Sales Representative' ,
E = 'Ms.' ,
F = date([1992,5,1],[0,0,0]) ,
G = '507 - 20th Ave. E.~M~JApt. 2A' ,
H = 'Seattle' ,
I = 'WA' ,
J = 98122 ,
K = 'USA' ,
L = '(206) 555-9857' ,
M = 5467 ,
N = '$address$'('019CC839:569') ,
O = 'Education includes a BA in psychology from Colorado
State University in 1970.  She also completed "The Art of
the Cold Call."  Nancy is a member of Toastmasters
International.' ,
P = 2 ;
```

Note: a single digit integer within a Prolog *date/2* term is always preceded by a zero when converted into SQL syntax.

From DBMS to Prolog

Type conversions are performed to conform to ***WIN-PROLOG*** as this is the programming side of the interface. Anything that can be interpreted by ***WIN-PROLOG*** as an integer is converted to an integer. The same is true of floating point numbers and unquoted atoms.

Beware: A value of , say, '10' in a column of type text will arrive at the ***WIN-PROLOG*** end as the integer 10. The value 10 in a column of type integer will arrive at the ***WIN-PROLOG*** still as the integer 10. There is no way at the ***WIN-PROLOG*** end to tell the difference except to look at the *data_dictionary/5* fact for the column to ascertain the original datatype.

Long binary values are returned to Prolog as addresses and are given the structure '$address$'(Address) where Address is a pointer to the address to which the value has been projected in memory. You would probably want to pass this address out to a foreign language function for processing.

There is a supplied predicate, ***db_get_byte_at_address/3***, for obtaining the values of bytes offset from the given address which can be used to interrogate long values from Prolog.

Dates are returned as the date structure defined in the previous section. Anything else is converted to a quoted atom, unless the value is too long for a Prolog atom, in which case it is converted to an LPA string.

The following example taken from the example database shows some of these conversions:

```
?- db_attach( emp, 'EMP' ).<return>

?- emp( Empno, Ename, Job, Mgr, Hdate, Sal, Comm, Dept ).
<return>
Empno = 7658,
Ename = 'CHAN',
Job = 'ANALYST',
Mgr = 7566,
Hdate = date([1982,3,5],[]),
Sal = 3450,
Comm = '$null$',
Dept = 20
```

Non-defined fields in database tables are converted to the Prolog atom '$null$'. Null fields may be selected by using the atom '$null$'.

It is possible to redefine how nulls are returned to Prolog by redefining the Prolog database entry *'db_null/1'*. This is set by default to '$null$':

```
?- db_null( X ). <return>
X = '$null$'

?- retract( db_null(_) ). <return>
yes

?- assert( db_null('$NULL$') ). <return>
yes


?- listing(db_null). <return>
% db_null/1
db_null( '$NULL$' ).
yes
```

If you require constants to match with database null values then *db_match_null(on)* should replace *db_match_null(off)* in the database which is provided by default. This can be achieved by using ***db_flag/2***:

```
?- db_match_null( X ). <return>
X = off

?- db_flag( match_db_null, Old, on ). <return>
Old = off
```

The predicate, ***db_flag/2***, is mentioned in detail in section 5.8.

Joins

Joins maybe effected in an identical fashion to Prolog, i.e. by having subgoals (corresponding to tables) which have a common variable. For example, to join the **EMP** table to the **DEPT** table using the common **DEPTNO** field, make the query:

```
?- emp( Empno, Ename, Job, Mgr, Hdate, Sal, Comm, Dept ),
dept( Dept, Dname, Location ).<return>
```

Relational Operators

Queries of the form:

```
?- dept( Dno, Dept, Loc ), Dno > 20.<return>
```

may be optimised by including the condition inside the table access goal in the following manner:

```
?- dept( op(Dno > 20), Dept, Loc ).<return>
```

This allows the DBMS to handle the condition, which improves the efficiency of the access.

The usual arithmetic operators are available (<, >, >=, =<) as well as the string operators, + and -, for begins with and occurs respectively.

SQL (Use in SQL query)	Prolog (Use in op/1 term)
=	=
<>	\=
<	<
>	>
<=	=<
>=	>=

For example, to retrieve all the departments based in a location starting with 'NE', the Prolog goal is:

```
?- dept( Dno, Dname, op(Loc+'NE') ).<return>
```

For example, to retrieve all the shippers with an ID less than or equal to 2 and which have the phrase 'Pack' within their name, the Prolog goal is:

```
?- shippers( op(A=<2), op(B-'Pack'), C ).<return>
SQL> SELECT ShipperID,CompanyName,Phone FROM Shippers
```

```
WHERE (ShipperID<=2 OR ShipperID IS NULL) AND
(CompanyName LIKE '%Pack%' OR CompanyName IS NULL)
A = 2 ,
B = 'United Package' ,
C = '(503) 555-3199' ;

no
```

Note: The *op/1* notation does not support the '<column> BETWEEN <value1> AND <value2>' SQL syntax.

Programmatically Generating An op/1 Term

An *op/1* term can be programmatically generated by using *~>/2*, *read/1* and *<~/2*:

```
?- ( write(`op(`), write(DummyVariable),
write(` `), write(`<`), write(` `), write(3),
write(`). `) )~> OpTermAsString.
Y = _ ,
OpTermAsString = `op(_37732 < 3). `

?- read(X) <~ `op(Y < 3). `. <return>
X = op(_1108 < 3)
```

Reading An op/1 Term

Should an *op/1* term be returned to your program, the value from the data source can be extracted using *=../2* as follows:

```
?- op(2 < 3) =.. [_,Argument], Argument =.. [_,Value,_].
<return>
Argument = 2 < 3 ,
Value = 2
```

Alternative access to DBMS tables

Sometimes the ***db_attach*** method of access may be inconvenient, such as when we do not know or care what the arity of a table is. In these situations it can be a lot more convenient to treat the record as a single structure. The ***db_tuple*** predicate will take the table name and return a tuple as a list comprised of its fields, and will backtrack to retrieve the next record upon failure. For example (for the default HDBC):

```
?- db_tuple( 'Shippers', R ).<return>
```

will return:

```
SQL> SELECT * FROM Shippers
R = [1,'Speedy Express','(503) 555-9831'] ;

R = [2,'United Package','(503) 555-3199'] ;

R = [3,'Federal Shipping','(503) 555-9931'] ;

no
```

The second argument will be used for building the query where it is partially or fully instantiated. The returned record is used to instantiate any variables in the list. For example:

```
?- db_tuple( 'DEPT', [A,'ACCOUNTING',C] ).<return>
```

causes the SQL query:

SELECT DEPTNO,LOC FROM DEPT WHERE DNAME='ACCOUNTING'

to be issued. The returned values for DEPTNO and LOC are used to instantiate variables A and C.

There is no need for a ***db_attach*** to be executed before using ***db_tuple*** on a table.

An augmentation string may be specified as an extra argument in argument position two, as in the example:

```
?- db_tuple( 'DEPT', `(dept.ndx/USE)`, R ).<return>
```

The augmentation string will be appended to the SQL for the generated query.

If you wish to supply the HDBC value then you must use the 4 argument version of ***db_tuple***:

```
?- db_tuple( 1, 'shippers', ``, Rec ).<return>
SQL> SELECT * FROM shippers
Rec = [1,'Speedy Express','(503) 555-9831'] ;

Rec = [2,'United Package','(503) 555-3199'] ;

Rec = [3,'Federal Shipping','(503) 555-9931'] ;

no
```

It is also possible to perform any SQL query using the ***db_sql_select/3*** predicate. This takes the HDBC atom as its first argument, an SQL string as its second and a variable or a list of variables as the third. If the third argument is a list it should be of a length corresponding to the number of field selections in the query, or given in head and tail form. Upon the SQL query being satisfied the third argument is instantiated to a list of

the field values corresponding to the SELECT part of the SQL statement. The first argument can be omitted if the default connection is to be used. For example:

```
?- db_attach(customers,'Customers'). <return>
?- db_sql_select(`select count(*) from customers`,[C]).
<return>
SQL> select count(*) from customers
C = 91 ;

?- db_sql_select( 'select ename, empno from emp', L
).<return>
L=['CHAN', 7658]

?- db_sql_select( 'select ename, empno from emp', [A,B]
).<return>
A='CHAN'
B=7658
```

and joining EMP and DEPT tables:

```
?- db_sql_select( 'SELECT ENAME,LOC FROM EMP,DEPT WHERE
EMP.DEPTNO=DEPT.DEPTNO ORDER BY ENAME', R ).<return>
R = ['ADAMS','DALLAS'];
R = ['ALLEN','CHICAGO'];
R = etc.,
```

Note that constants in the list are **NOT** acceptable. This is a low level access routine and unification on the arguments is not taking place. If you need any such unification then define a higher order predicate that forces it, for example:

```
unify_sql( HDBC, Q, R ):-
   db_sql_select( HDBC, Q, R1 ),
   R = R1.
```

though it will be far more efficient for you to build the full query every time and avoid unification.

Building An SQL Query Up As a List

You may want to instantiate variables in the SQL query prior to submission; following is a generic predicate that will accept an SQL query as a list:

```
db_sql_select_list( QueryList, Result ) :-
  list( QueryList ),
  !,
  ( forall( member( Bit, QueryList ),
            ( write( Bit ),
              write( ` ` )
            )
          )
```

```
  ) ~> QueryString,
  db_sql_select( QueryString, Result ).

?- db_sql_select_list( [select,'*',from,customers],
Result ). <return>

?- Table = customers, db_sql_select_list( [ select, '*',
from, Table ], Result ). <return>

?- db_sql_select_list( [ 'SELECT * FROM CUSTOMERS WHERE
Country =', `'Germany'` ], Result ). <return>
SQL> SELECT * FROM CUSTOMERS WHERE Country = 'Germany'
```

Cursor based access

It is possible to access tuples based on any selection query procedurally, i.e. there is no backtracking involved in accessing the next record. To start such a query use:

```
?- db_open_cursor( 'SELECT * FROM shippers', C ).<return>
C = 1
```

Tuples for the given query can then be accessed by using:

```
?- db_cursor_tuple( 1, R ).<return>
R = [1,'Speedy Express','(503) 555-9831']

?- db_cursor_tuple( 1, R ).<return>
R = [2,'United Package','(503) 555-3199']

?- db_cursor_tuple( 1, R ).<return>
R = [3,'Federal Shipping','(503) 555-9931']

?- db_cursor_tuple( 1, R ).<return>
no
```

where 1 is the cursor number. This will succeed and instantiate R to the list of field values satisfying the query. It will ultimately fail when there are no more tuples in which case the cursor will be automatically closed. If you wish to terminate the access before retrieving all matching tuples, use:

```
?- db_close_cursor( 1 ).<return>
yes
```

where 1 is the cursor number.

In certain DataBase Management Systems, such as Microsoft Access 97, it is not possible to have two cursors open at the same time.

Long fields

ODBC recognises two kinds of long field, binary and text (long varchar). Long text fields are returned as strings to Prolog. The maximum number of characters is set by default to 4096 bytes. This can be changed by using ***db_set_max_blob_size/1***. The maximum number of bytes that can be projected for a single tuple is 64K. Long values are allocated out of this memory area. So beware, you cannot project out three fields each of which is potentially 32K from a single query.

Long binary fields, such as is used for bit maps are returned to Prolog as a structure which gives the address in memory of where they have been copied. An example of address structure is:

'$address$'('27A7:023B:571')

The first 9 characters are the address of the binary field as obtained by using the "%p" formatting option within 'C'. The number after the second colon indicates an array offset and is used internally by ProData.

Bytes can be retrieved from a particular address by using ***db_get_byte_at_address/3***. For example to get the byte value at the above given address, use:

```
?- db_get_byte_at_address( '$address$'('27A7:023B:571'),
0, V ).<return>
```

The byte at the next address can be obtained by changing the 0 to a 1, and so on.

Chapter 4 - DATA DICTIONARY

Data sources

A list of available data sources can be obtained with:

```
?- db_show_schema( sources ).<return>
MS Access 97 Database
dBASE Files
Excel Files
FoxPro Files
Text Files
nwind
yes
```

The list may be instantiated to a variable with ***db_get_schema/2***. For example:

```
?- one( db_get_schema( sources, L ) ). <return>
L = [`nwind`,`MS Access 97 Database`,`dBASE Files`,`Excel
Files`,`FoxPro Files`,`Text Files`]
```

You do not need to be connected to a data source to use either of these predicates and it is not possible or meaningful to specify an HDBC.

Database tables

You can display the accessible table names and table owners/directories for those tables that can be accessed on the default connection using:

```
?- db_show_schema( accessible ).<return>
$null$.Categories
$null$.Customers
$null$.Employees
$null$.Order Details
$null$.Orders
$null$.Products
$null$.Shippers
$null$.Suppliers
yes
```

will display the names of all accessible tables along with their owners. A particular HDBC may be specified by providing it as the first argument. For example:

```
?- db_show_schema( 1, accessible ).<return>
$null$.Categories
$null$.Customers
$null$.Employees
```

```
$null$.Order Details
$null$.Orders
$null$.Products
$null$.Shippers
$null$.Suppliers
yes
```

where 1 is a valid HDBC number as returned by ***db_connect/2***.

The list may be instantiated to a variable by using ***db_get_schema/[2,3]***. For example:

```
?- db_get_schema( accessible, L ).<return>
L = ['$null$.Categories', '$null$.Customers',
'$null$.Employees', '$null$.Order Details',
'$null$.Orders', '$null$.Products', '$null$.Shippers',
'$null$.Suppliers']
```

A particular connection (HDBC) may be specified by providing it as the first argument to ***db_get_schema/3***.

If you want to get rid of the '$null$.' prefix, the following code may help:

```
db_get_accessible_schema( Tables2 ) :-
   db_get_schema( accessible, Tables ),
   findall( Table3,
            ( member( Table, Tables ),
              atom_string( Table, Table2 ),
              replace(`$null$.`,``)~>Table3 <~Table2
            ),
            Tables2
          ).

replace( From, To ) :-
  find( From, 2, Found ),
  Found = ``,
  !.

replace( From, To ) :-
  write( To ),
  replace( From, To ).

?- db_get_accessible_schema( Tables ). <return>
Tables = [`Customers`, `Employees`, `Order Details`,
`Orders`, `Products`, `Shippers`, `Suppliers`,
`Categories`]
```

The predicate ***db_show_schema/2*** when used with the atom 'accessible' will usually produce a long list as it will include all the data dictionary tables and views. If you wish to see just those tables that belong to you then use the same predicate with the atom 'user'. Its form is:

```
?- db_show_schema( user ).<return>
Categories
Customers
Employees
Order Details
Orders
Products
Shippers
Suppliers
yes
```

There is an equivalent ***db_get_schema/[2,3]*** predicate (the HDBC can be optionally provided as for previous predicates in this section):

```
?- db_get_schema( user, R ).<return>
R = ['Categories', 'Customers', 'Employees', 'Order
Details', 'Orders', 'Products', 'Shippers', 'Suppliers']
```

Note: If the datasource does not contain any tables, ***db_get_schema/[2,3]*** will fail whereas ***db_show_schema/[1,2]*** with succeed.

Table arity

The number of columns in (arity of) a particular table can be found in the following way:

```
?- db_show_schema( arity(shippers) ).<return>
3
yes
```

There is an equivalent ***db_get_schema/[2,3]*** predicate (the HDBC can be optionally provided as for previous predicates in this section):

```
?- db_get_schema( 1, arity(shippers), Arity ).<return>
Arity = 3
```

Table columns

Column names for a table can be found using ***db_show_schema/[1,2]***; this should return the column names in the order they appear in the table:

```
?- db_show_schema( columns(orders) ). <return>
OrderID
CustomerID
EmployeeID
OrderDate
RequiredDate
ShippedDate
ShipVia
Freight
```

```
ShipName
ShipAddress
ShipCity
ShipRegion
ShipPostalCode
ShipCountry
yes
```

There is an equivalent ***db_get_schema/[2,3]*** predicate (the HDBC can be optionally provided as for previous predicates in this section):

```
?- db_get_schema( columns(orders), C ). <return>
C = ['ShipCity', 'CustomerID', 'ShipRegion',
'EmployeeID', 'ShipPostalCode', 'OrderDate',
'ShipCountry', 'RequiredDate', 'ShippedDate', 'ShipVia',
'Freight', 'ShipName', 'ShipAddress', 'OrderID']
```

The predicate, ***db_get_schema/[2,3]***, relies upon the ***data_dictionary/5*** facts being asserted into memory; these are asserted automatically when you, for example, attach to a table.

The following program converts the output from ***db_show_schema/1*** into a list:

```
my_db_show_schema( Term, List ) :-
  one( db_show_schema(Term) ~> String ),
  string_list( String, List ).

string_list( String, List ) :-
  string_list2( [], ReversedList ) <~ String,
  reverse( ReversedList, List ).

string_list2( Dummy, List ) :-
  find( `~M~J`, 3, Found ) ~> Element,
  (  Found = ``
  -> (  Element = ``
     -> List = Dummy
     ;  List = [Element|Dummy]
     )
  ;  (  Element = ``
     -> string_list2( Dummy,           List )
     ;  string_list2( [Element|Dummy], List )
     )
  ).

?- my_db_show_schema( columns(orders), List ). <return>
List = [`OrderID`, `CustomerID`, `EmployeeID`,
`OrderDate`, `RequiredDate`, `ShippedDate`, `ShipVia`,
`Freight`, `ShipName`, `ShipAddress`, `ShipCity`,
`ShipRegion`, `ShipPostalCode`, `ShipCountry`]
```

Table tuples

Tuples may be listed for a particular table using:

```
?- db_show_schema( tuples(shippers) ).<return>
SQL> SELECT * FROM shippers
shippers(1,Speedy Express,(503) 555-9831)
shippers(2,United Package,(503) 555-3199)
shippers(3,Federal Shipping,(503) 555-9931)
yes
```

An HDBC may be optionally specified but there is no ***db_get_schema*** version of this function as it may cause severe memory problems due to the number of possible records that can be retrieved.

Data dictionary tables

Data dictionary tables may be attached to Prolog predicates in the same way that a user's tables may be, that is by using ***db_attach***. An example relevant to Oracle is:

```
?- db_attach( access, ['ACCESSIBLE_COLUMNS'] ).<return>
```

connects to the Oracle data dictionary table that describes the columns for a user's accessible tables. The average user will not need to concern himself with the contents of the data dictionary, but for anyone who is providing a customised front-end to the DBMS accessing this information will be essential.

You may not be able to attach to certain data dictionary tables where these are not genuine tables or you are denied access to them. This is database dependant.

Chapter 5 - OTHER DATABASE OPERATIONS

Creating tables and indexes

A database table can be created from within Prolog by using the ***db_create_table*** predicate. This requires the table name and fields specification to be passed as parameters. The fields specification is a string which is identical to that used by SQL. Examples are:

```
?- db_create_table( 'DEPT', 'DEPTNO NUMERIC(2),DNAME
CHAR(14),LOC CHAR(13)' ).<return>

?- db_create_table( 'DEPT', 'DEPTNO NUMBER,DNAME
CHAR(14),LOC CHAR(13)' ).<return>
SQL> CREATE TABLE DEPT(DEPTNO NUMBER,DNAME CHAR(14),LOC
CHAR(13))
yes
```

Make sure that the fields specification parameters are those expected by your database management system.

An index can be created for any column or set of columns using ***db_create_index***. This requires the table name, index name, one of the atoms 'unique' or 'non_unique', and a column (or columns) to be given as parameters, thus:

```
?- db_create_index( 'EMP', 'EMP_DEPTNO', non_unique,
'DEPTNO').<return>
```

creates an index on a single column, **DEPTNO**, which may have repeated values in different records.

```
?- db_create_index( 'EMP', 'EMP_KEY', unique,
'DEPTNO,EMPNO' ).<return>
```

creates an index based on the concatenation of the **DEPTNO** and **EMPNO** columns. The **unique** flag forces the constraint that there must be no repeated occurrences of the same DEPTNO and EMPNO values.

Deleting tables, views and indexes

To delete a table use the ***db_delete_table/[1,2]*** predicate giving the table name as the argument. For example:

```
?- db_delete_table( 'DEPT' ).<return>
SQL> DROP TABLE DEPT
yes
```

To delete a view use the ***db_delete_view/[1,2]*** predicate giving the view specification as the argument. This is not available for those databases which do not support views. For example:

```
?- db_delete_view( 'EMPO' ).<return>
```

To delete an index use ***db_delete_index/[1,2]*** giving the index specification as the argument. For example:

```
?- db_delete_index( 'EMP_KEY' ).<return>
SQL> DROP INDEX EMP_KEY

yes
```

Some databases may want the index name to be qualified by the table name. For example, with the Q+E dBASE IV driver the table name needs to be prefixed to the index name, as in:

```
?- db_delete_index( 'EMP.EMP_KEY' ).<return>
SQL> DROP INDEX EMP.EMP_KEY
yes
```

Adding and deleting records

Rows may be added (inserted) into an existing table by using the ***db_add_record*** predicate. The arguments are the table name and a list composed of the field values. For example:

```
?- db_add_record( 'Shippers', [4,'Keylink Computers',
'01926 850909'] ).<return>
SQL> INSERT INTO Shippers VALUES(4, 'Keylink Computers',
'01926 850909')
yes
```

To insert a record with a NULL value in a column, use the atom '$null$' as the value for the column(s). To insert a record containing a date, use the date structure described earlier. LPA strings may be used in place of atoms.

```
?- db_add_record('Shippers', [5,`LPA`,'$null$']).<return>
SQL> INSERT INTO Shippers VALUES(5,'LPA',NULL)
yes
```

If you do not wish a particular string to be quoted when using ***db_add_record/2*** then assert *db_do_not_quote(atom)* into Prolog's database: For example, I may not want the string 'sysdate' to be quoted as this is a keyword for some particular database, I would do:

```
?- assert( db_do_not_quote(sysdate) ).<return>
yes
```

Rows may be deleted using the ***db_delete_record*** predicate. This has the same arguments as ***db_add_record***. For example:

```
?- db_delete_record( 'Shippers', [4|_] ).<return>
SQL> DELETE FROM Shippers WHERE (ShipperID=4)
yes

?- db_delete_record( 'Shippers', [_,'LPA',_] ).<return>
SQL> DELETE FROM Shippers WHERE (CompanyName='LPA')
yes

?- db_delete_record( 1, 'Shippers', [_,'LPA'|_] ).
<return>
SQL> DELETE FROM Shippers WHERE (CompanyName='LPA')
yes
```

Notice that not every field needs to be instantiated.

Handling Prolog Lists

It is not possible to add a record which contains a Prolog list as one of its field values to a data source. The following code might help if you need to convert a real Prolog list into a string representation of a Prolog list or vice versa:

```
?- List = [this,is,a,prolog,list], write( List ) ~>
String. <enter>
List = [this,is,a,prolog,list] ,
String = `[this,is,a,prolog,list]`
?- String1=`[this,is,a,prolog,list]`, cat([String1,`. `],
String2, _ ), eread( List ) <~ String2. <return>
String1 = `[this,is,a,prolog,list]` ,
```

```
String2 = `[this,is,a,prolog,list]. ` ,
List = [this,is,a,prolog,list]
```

Other SQL commands

There are other SQL commands you may wish to execute which have not had a Prolog predicate specifically defined for them. Such functions as enroling new users do not have specific Prolog predicates defined within this interface. However, they can still be executed from Prolog using ***db_sql***, which takes the DBMS and a string as its arguments. This string must correspond to a valid SQL statement which does not involve data retrieval. If you wish to retrieve data with an SQL query from Prolog then see the description of ***db_sql_select*** in chapter 3.

An example relevant to Oracle is: to enrol the user **fred** with password **bloggs**, with the **connect only** privilege, use the following goal:

```
?- db_sql( 'grant connect to fred identified by bloggs'
).<return>
SQL> grant connect to fred identified by bloggs
yes
```

NOTE: All the above predicates exist in a form with an extra argument should you wish to specify the HDBC directly; the HDBC is then the first argument.

Seeing the SQL for a generated query

If you wish to see on the screen the SQL that is generated by the interface use the predicate ***db_flag/3*** with the show_sql atom. For example:

```
?- db_flag( show_sql, O, on ).<return>
O = on

?- db_flag( show_sql, _, on ).<return>
yes
```

switches the show SQL facility on, and O will be instantiated to its previous setting, while:

```
?- db_flag( show_sql, O, off ).<return>
O = on

?- db_flag( show_sql, _, off ).<return>
yes
```

switches the show SQL facility off. The default behaviour is to show the generated SQL.

```
?- db_flag( show_sql, _, S ).<return>
S = on
```

Seeing DBMS error messages

If you wish to see on the screen any error messages that might occur due to Prodata and database access errors while using Prodata use the predicate ***db_flag/3*** with the *show_db_error* atom. For example:

```
?- db_flag( show_db_error, O, on ).<return>
O = on
```

switches the error reporting facility on, and O will be instantiated to its previous setting, while:

```
?- db_flag( show_db_error, O, off ).<return>
O = on
```

switches the error reporting facility off. The default behaviour is to show the error messages.

```
?- db_flag( show_db_error, _, S ).<return>
S = on
```

When error messages are reported an attempt is made to indicate the goal that caused the problem. This will not always be a goal that you called directly but will be one of the documented calls which is being used to answer your query.

If you do not wish error messages to appear directly, but you still wish to be able to retrieve error messages, then have *show_db_error* set to off and use:

```
?- one( db_flag(record_db_error,O,on) ).<return>
O = off
```

This will cause a fact of the form:

db_error_message(Goal, Errno, ErrMess).

to be asserted into the Prolog database each time an error is encountered.

Setting default connection

The default database connection (HDBC) can be set using ***db_flag/3*** with the default_connection argument. For example:

```
?- db_flag( default_connection, O, 2 ).<return>
```

sets the default HDBC to 2. 'O 'will be instantiated to the previous value for the default ODBC. A value of zero indicates no default is/was set. The value supplied must be a legal HDBC, that is it must have been returned by ***db_connect***.

Matching constants with nulls

If you require constants to match with database null fields use:

```
?- db_flag( match_db_null, O, on ).<return>
O = off
```

'O' will be set to the current setting. Using 'off' in place of 'on' will turn the matching off.

```
?- db_flag( match_db_null, O, off ).<return>
O = on
```

Chapter 6 - TRANSACTION MANAGEMENT

Transactions

Normally from Prodata, any changes to the database will be committed (i.e. take permanent effect) as each change happens. This is often referred to as 'autocommit'.

In order to provide the user with further control over this process the transaction management predicates are provided:

db_begintran/[0,1]: marks the start of a transaction on a database connection. (ODBC has no direct implementation of this, the interface implements this by turning off autocommit).

db_commit/[0,1]: commits all pending changes which have been made since issuing the ***db_begintran***. This is equivalent to the SQL command **COMMIT**. If successful, autocommit is turned back on.

db_rollback/[0,1]: causes all pending changes made since issuing the ***db_begintran*** to be discarded. This is equivalent to the SQL command **ROLLBACK**. If successful, autocommit is turned back on.

All three transaction predicates can be used with no argument, in which case the default database connection will be used. Or they may be used in the form where they take a single argument which should be instantiated to a legal database connection (HDBC).

From the point at which ***db_begintran*** is executed no updates are committed until ***db_commit*** is executed at which point the changes are made permanent and can no longer be rolled back. At any point after ***db_begintran*** has been executed any changes may be rolled back by using ***db_rollback***.

```
transaction :-
   db_begintran,
   db_add_record( 'Shippers', [4,'Keylink Computers',
'01926 850909'] ),
   db_show_schema( tuples('Shippers') ),
   db_rollback,
   db_show_schema( tuples('Shippers') ).

?- transaction.<return>
SQL> INSERT INTO Shippers VALUES(4,'Keylink
Computers','01926 850909')
SQL> SELECT * FROM Shippers
Shippers(1,Speedy Express,(503) 555-9831)
Shippers(2,United Package,(503) 555-3199)
```

```
Shippers(3,Federal Shipping,(503) 555-9931)
Shippers(4,Keylink Computers,01926 850909)
SQL> SELECT * FROM Shippers
Shippers(1,Speedy Express,(503) 555-9831)
Shippers(2,United Package,(503) 555-3199)
Shippers(3,Federal Shipping,(503) 555-9931)
yes
```

Chapter 7 – SQL EXAMPLES

The predicate, *db_sql_select_list/2*, is defined earlier.

This example asks for the number of distinct values in the Region column of the Employees table:

```
?- Table='Employees',Cols='Region',db_sql_select_list(
['select COUNT(', Cols, ') from' ,Table],Res),
write(Res), nl, fail. <return>
SQL> select DISTINCT COUNT( Region ) from Employees
[5]
```

This example asks for the number of occurrences of 'OR' in the ShipRegion column of the Orders table:

```
?- db_sql_select( `select count(shipregion) from orders
where shipregion='OR'`, P). <return>
SQL> select count(shipregion) from orders where
shipregion='OR'
P = [28] ;
```

This example asks for all the distinct values and the occurrence of each in the ShipRegion column of the Orders table:

```
?- db_sql_select( 'select ShipRegion, COUNT( ShipRegion )
from Orders Group by ShipRegion', P). <return>
SQL> select ShipRegion, COUNT( ShipRegion ) from Orders
Group by ShipRegion
P = ['$null$',0] ;
P = ['AK',10] ;
P = ['BC',17] ;
P = ['CA',4] ;
P = ['Co. Cork',19] ;
P = ['DF',2] ;
P = ['Essex',13] ;
P = ['ID',31] ;
P = ['Isle of Wight',10] ;
P = ['Lara',14] ;
P = ['MT',3] ;
P = ['NM',18] ;
P = ['Nueva Esparta',12] ;
P = ['OR',28] ;
P = ['Québec',13] ;
P = ['RJ',34] ;
P = ['SP',49] ;
P = ['Táchira',18] ;
P = ['WA',19] ;
P = ['WY',9] ;
```

This example asks for all the distinct values and the occurrence of each in the ShipRegion column of the Orders table where the value in the ShipVia column is 3; Count(*) is also output and you can see how this differs when the value, '$null$', is encountered:

```
?- db_sql_select( 'select ShipRegion, COUNT( ShipRegion
), COUNT(*) from Orders where shipvia = 3 Group by
ShipRegion', P). <return>
SQL> select ShipRegion, COUNT( ShipRegion ), COUNT(*)
from Orders where shipvia = 3 Group by ShipRegion
P = ['$null$',0,157] ;
P = ['AK',5,5] ;
P = ['BC',11,11] ;
P = ['Co. Cork',6,6] ;
P = ['DF',2,2] ;
P = ['Essex',4,4] ;
P = ['ID',11,11] ;
P = ['Isle of Wight',2,2] ;
P = ['Lara',5,5] ;
P = ['MT',1,1] ;
P = ['NM',8,8] ;
P = ['Nueva Esparta',3,3] ;
P = ['OR',7,7] ;
P = ['Québec',5,5] ;
P = ['RJ',5,5] ;
P = ['SP',12,12] ;
P = ['Táchira',3,3]
```

This example asks for the standard deviation, minimum, maximum and average of the values in the UnitPrice column of the Order Details table:

```
?- db_sql_select( 'select stdev(unitprice),
min(unitprice), max(unitprice), avg(unitprice) from
"Order Details"', P). <return>
SQL> select stdev(unitprice), min(unitprice),
max(unitprice), avg(unitprice) from "Order Details"
P = [29.827417887501,2,263.5]
```

This example asks for the sum, a count, standard deviation, minimum * maximum and average of the values in the UnitPrice column of the Order Details table:

```
?- db_sql_select( 'select sum(unitprice),
count(unitprice), stdev(unitprice),
min(unitprice)*max(unitprice), avg(unitprice) from "Order
Details"', P). <return>
SQL> select sum(unitprice), count(unitprice),
stdev(unitprice), min(unitprice), max(unitprice),
avg(unitprice) from "Order Details"
P = [56500.91,2155,29.827417887501,2,263.5,26.2185]
```

This example asks for the top seven values in the ShipRegion column of the Orders table with the highest occurrence; a count of the occurrences of each is also returned:

```
?- db_sql_select( 'select top 7 ShipRegion, COUNT(
ShipRegion ) from Orders Group by ShipRegion Order by
COUNT( ShipRegion) desc', P). <return>
SQL> select top 7 ShipRegion, COUNT( ShipRegion ) from
Orders Group by ShipRegion Order by COUNT( ShipRegion)
desc
P = ['SP',49] ;
P = ['RJ',34] ;
P = ['ID',31] ;
P = ['OR',28] ;
P = ['WA',19] ;
P = ['Co. Cork',19] ;
P = ['Táchira',18] ;
P = ['NM',18] ;
no
```

This example asks for the variance, a count, standard deviation, minimum * maximum and average of the values in the UnitPrice column of the Order Details table:

```
?- db_sql_select( 'select var(unitprice),
count(unitprice), stdev(unitprice),
min(unitprice)*max(unitprice), avg(unitprice) from "Order
Details"', P). <return>
SQL> select var(unitprice), count(unitprice),
stdev(unitprice), min(unitprice)*max(unitprice),
avg(unitprice) from "Order Details"
P = [889.674857835614,2155,29.827417887501,527,26.2185]
```

This example asks for the varp, a count, standard deviation, minimum * maximum and average of the values in the UnitPrice column of the Order Details table:

```
?- db_sql_select( 'select varp(unitprice),
count(unitprice), stdev(unitprice),
min(unitprice)*max(unitprice), avg(unitprice) from "Order
Details"', P). <return>
SQL> select varp(unitprice), count(unitprice),
stdev(unitprice), min(unitprice)*max(unitprice),
avg(unitprice) from "Order Details"
P = [889.262015674205,2155,29.827417887501,527,26.2185]
```

This example asks for the distinct values in all the columns of the Employees table:

```
?- Table='Employees', db_get_schema(columns(Table), L),
member(Cols,L), db_sql_select_list( ['select DISTINCT',
Cols, 'from' ,Table],[Res]), write(Res), nl, fail.
<return>
SQL> select DISTINCT Title from Employees
Inside Sales Coordinator
Sales Manager
Sales Representative
Vice President, Sales
SQL> select DISTINCT Extension from Employees
2344
3355
```

```
3453
3457
428
452
465
5176
5467
SQL> select DISTINCT TitleOfCourtesy from Employees

Dr.
Mr.
Mrs.
Ms.
SQL> select DISTINCT BirthDate from Employees
date([1937,9,19],[0,0,0])
date([1948,12,8],[0,0,0])
date([1952,2,19],[0,0,0])
date([1955,3,4],[0,0,0])
date([1958,1,9],[0,0,0])
date([1960,5,29],[0,0,0])
date([1963,7,2],[0,0,0])
date([1963,8,30],[0,0,0])
date([1966,1,27],[0,0,0])
SQL> select DISTINCT HireDate from Employees
date([1992,4,1],[0,0,0])
date([1992,5,1],[0,0,0])
date([1992,8,14],[0,0,0])
date([1993,5,3],[0,0,0])
date([1993,10,17],[0,0,0])
date([1994,1,2],[0,0,0])
date([1994,3,5],[0,0,0])
date([1994,11,15],[0,0,0])
SQL> select DISTINCT ReportsTo from Employees
$null$
2
5
SQL> select DISTINCT Address from Employees
14 Garrett Hill
4110 Old Redmond Rd.
4726 - 11th Ave. N.E.
507 - 20th Ave. E.
Apt. 2A
7 Houndstooth Rd.
722 Moss Bay Blvd.
908 W. Capital Way
Coventry House
Miner Rd.
Edgeham Hollow
Winchester Way
SQL> select DISTINCT City from Employees
Kirkland
London
Redmond
Seattle
Tacoma
SQL> select DISTINCT Region from Employees
$null$
```

```
WA
SQL> select DISTINCT EmployeeID from Employees
1
2
3
4
5
6
7
8
9
SQL> select DISTINCT PostalCode from Employees
98033
98052
98105
98122
98401
EC2 7JR
RG1 9SP
SW1 8JR
WG2 7LT
SQL> select DISTINCT LastName from Employees
Buchanan
Callahan
Davolio
Dodsworth
Fuller
King
Leverling
Peacock
Suyama
SQL> select DISTINCT Country from Employees
UK
USA
SQL> select DISTINCT FirstName from Employees
Andrew
Anne
Janet
Laura
Margaret
Michael
Nancy
Robert
Steven
SQL> select DISTINCT HomePhone from Employees
(206) 555-1189
(206) 555-3412
(206) 555-8122
(206) 555-9482
(206) 555-9857
(71) 555-4444
(71) 555-4848
(71) 555-5598
(71) 555-7773
no
```

Chapter 8 – PRODATA AND STANDALONE APPLICATIONS

Following is an example of a standalone application using Prodata:

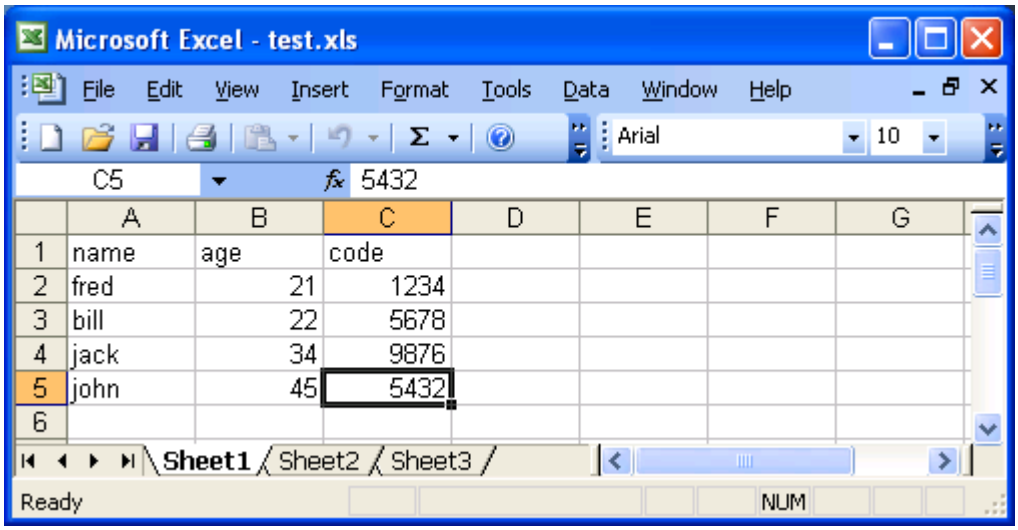
```
my_main_hook :-
  ensure_loaded( system(dblink) ),
  db_connect( northwind ),
  db_sql_select( 'SELECT * FROM CUSTOMERS', Record ),
  message_box( okcancel, Record, Response ),
  (  Response = ok
  -> fail
  ;  abort
  ).

my_main_hook :-
  abort.

my_abort_hook :-
  db_halt,
  halt.
```

If your standalone application needs to use Prodata, it will need to find LPADBW.DLL and DBLINK.PC. When testing within the ***WIN-PROLOG*** environment, the root and the SYSTEM directories are those of ***WIN-PROLOG***. When deploying a standalone application, it normally resides in its own directory away from ***WIN-PROLOG***; don't forget to copy DBLINK.PC from ***WIN-PROLOG***'s SYSTEM directory to the standalone application's SYSTEM directory and LPADBW.DLL from ***WIN-PROLOG***'s root directory to the standalone application's root directory.

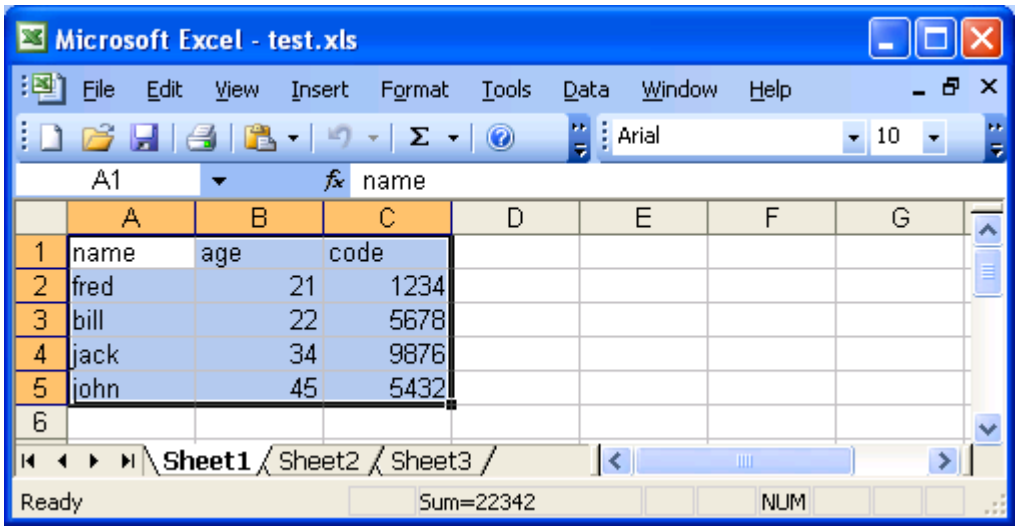
Chapter 9 – CONNECTING TO EXCEL FILES

This chapter shows you how to set up a 'table' in Excel and read it via ProData.

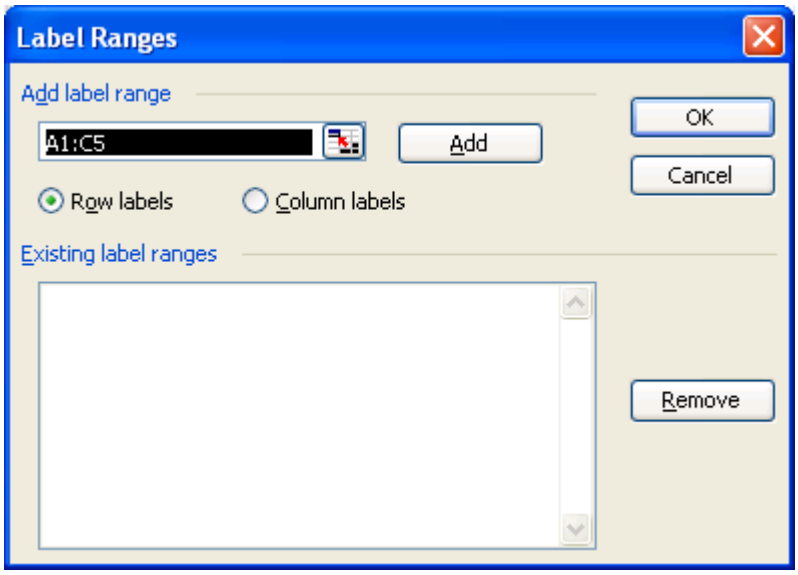
Firstly, create a simple spreadsheet in Excel:



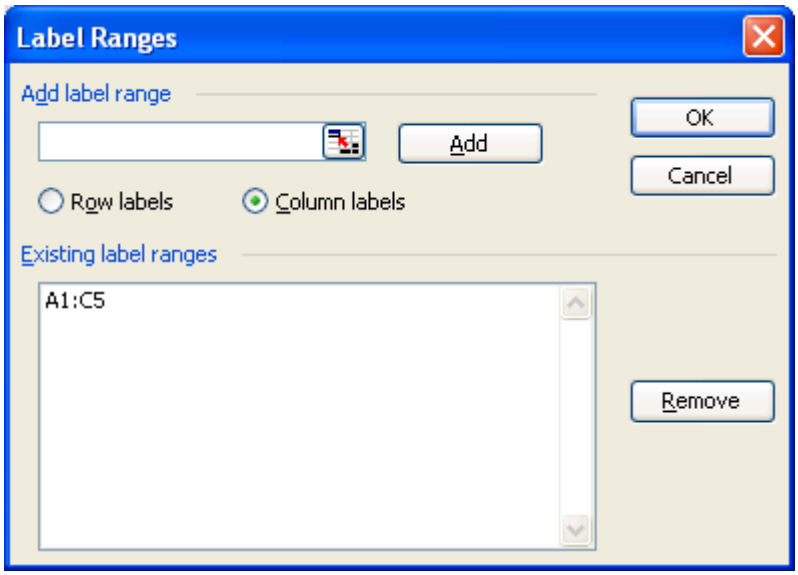
Block the area of the spreadsheet that you want to become a 'table'. We are going to specify the first row as the column headings:



Click on the Insert\Name\Label… menu option. The Label Ranges dialog will appear. The default name given for our 'table' is 'A1:C5'; this can be changed if required:



Select the 'Column labels' radio and click on the Add button. A 'table' called 'A1:C5' has been created.



Click on the OK button.

With Excel closed and the Excel spreadsheet set up as an ODBC data source, we can now read the data:

```
?- ensure_loaded( system(dblink) ). <enter>
yes

?- db_connect( my_excel_spreadsheet, HDBC ). <enter>
HDBC = 1

?- db_attach( my_excel_table, 'A1:C5' ). <enter>
yes

?- listing. <enter>
% my_excel_table/3
my_excel_table( A, B, C ) :-
   db_tuple( 1, 'A1:C5', ``, [A,B,C] ).

% data_dictionary/5
data_dictionary( 1, 'A1:C5', 3, code, 'NUMBER' ).
data_dictionary( 1, 'A1:C5', 2, age, 'NUMBER' ).
data_dictionary( 1, 'A1:C5', 1, name, 'CHAR' ).



?- my_excel_table( A, B, C ). <enter>
SQL> SELECT name,age,code FROM A1:C5
A = fred ,
B = 21 ,
C = 1234 ;

A = bill ,
B = 22 ,
C = 5678 ;

A = jack ,
B = 34 ,
C = 9876 ;

A = john ,
B = 45 ,
C = 5432 ;

no

?-
```

Chapter 10 - TROUBLE SHOOTING

Prolog Error

If you get the following error message when loading DBLINK.PC:

```
?- ensure_loaded(system(dblink)). <return>
ERROR – Failed to open Prodata DLL:lpadbw.dll
yes
```

it is because LPADBW.DLL cannot be found in either the *WIN-PROLOG* root directory or in a folder mentioned within your computer's PATH system variable.

If you were to execute **ensure_loaded(system(dblink))** again straight away, no error will be given but the goal still succeeds. Please ensure in your code that this predicate is executed only once and succeeds without an error message the very first time.

You could try redirecting any output from **ensure_loaded(system(dblink))** to a string and testing whether the string is empty or not:

```
?- ensure_loaded(system(dblink)) ~> Error. <return>
Error = `ERROR - Failed to open Prodata
DLL:lpadbw.dll~M~J`
```

Even if you were now to put LPADBW.DLL on the path and try executing **ensure_loaded(system(dblink))** again, you will find that although it still succeeds, errors will be returned later on:

```
?- db_connect(northwind). <enter>
! ----------------------------------------
! Error 30 : File Handling Error
! Goal     : winapi(('lpadbw.dll',c_db_connect),
[0,`northwind`],0,_3806)
```

The difference between a successful **ensure_loaded(system(dblink))** and an unsuccessful one is that on a successful one, you will have a **db_dll_handle/2** fact present in the Prolog database:

```
?- listing(db_dll_handle). <return>
% db_dll_handle/2
db_dll_handle( `lpadbw.dll`, 268435456 ).
Yes
```

As a result, a double check as to whether LPADBW.DLL has indeed been loaded might be to execute:

```
?- ensure_loaded(system(dblink)) ~> Error, Error =``.
<return>
```

or…

```
?- ensure_loaded(system(dblink)), db_dll_handle( _, _ ).
<return>
yes
```

Database Interface Error

Should you get a "Database interface error" whilst trying to connect to a data source with ***db_connect/[1,2]***, it may be because:

the data source has not been set up correctly in the ODBC Administrator (the ODBC Administrator's entry remains but the actual database file has been moved or deleted), or

it needs to be repaired (this can be done via the ODBC Administrator), or

the data source is password protected (the password can be passed in as a parameter within the ***db_connect/[1,2]*** predicate).

If you are using Prodata within a ProWeb or WebFlex application, has the data source been set up as a System DSN?

Just because Prodata says that a particular data source is available (via ***db_show_schema/1*** or ***db_get_schema/2***), it does not mean that the data source can actually be connected to.

The ODBC Administrator's Tracing Facility

The ODBC Administrator provides a tracing facility which allows you to create logs of the calls to ODBC drivers to aid you in debugging your applications. Note that these log files can become very large.

The Debugging Version of LPADBW.DLL

There are two versions of LPADBW.DLL. The debugging version, which is 35K in size, creates a log file, DEBUG.LOG, in the ***WIN-PROLOG*** root directory.

TRACE.PL

Should you need to debug your own code, TRACE.PL (in the ***WIN-PROLOG*** library directory) may prove useful; during tracing, you will need to press the <*spacebar*> to advance to the next goal):

```
51

?- ensure_loaded(library(trace)). <return>
yes
?- ?( ensure_loaded(system(dblink)) ). <return>
C ensure_loaded(system(dblink))
ERROR - Failed to open Prodata DLL:lpadbw.dll
S ensure_loaded(system(dblink))
yes
```

If you want to output to a file instead, try:

```
?- fcreate( mytrace, 'c:\mytrace.txt', -1,0,0 ). <return>
yes
?- ?( ensure_loaded(system(dblink)), mytrace ). <return>
```

You will need to press the *<space bar>* several times at this point.

```
yes
?- fclose(mytrace). <return>
yes
```

ODBC Drivers

There are many versions of Microsoft's ODBC drivers. The ODBC Data Source Administrator's 'Drivers' tab will tell you what set you have installed on your computer. Make sure you are using the correct set.

Chapter 11 - SUMMARY OF PREDICATES

Introduction

This is a description of the Prodata predicates as they are called from Prolog. Each synopsis is headed by a template which uses the mode annotations:

```
+    Input argument
-    Output argument
```

Where an argument is given as +HDBC, you should use the appropriate database connection number as detailed in the section: "Connecting to a DBMS" in Chapter 3. Where this argument is shown in square brackets, as in [+HDBC] then it may be omitted and the default connection will be used.

The predicates

---

db_add_record/2

To add a record to a database table

db_add_record(tname, fieldlist)

\+ **tname** table specification (list)

\+ **fieldlist** values to be inserted (list)

```
?- db_add_record( 'Shippers', [4,'Keylink
Computers','01926 850909'] ). <return>
SQL> INSERT INTO Shippers VALUES(4,'Keylink
Computers','01926 850909')
yes
```

---

db_add_record/3

To add a record to a database table

db_add_record(HDBC, tname, fieldlist)

\+ **HDBC** database system (atom)

\+ **tname** table specification (list)

\+ **fieldlist** values to be inserted (list)

---

db_attach/2

Connects a database table to a Prolog functor

db_attach(pfunc, tname)

\+ **pfunc** Prolog functor (atom)
\+ **tname** table name (atom or LPA string)

---

db_attach/3

Connects a database table to a Prolog functor

db_attach(pfunc, tname, aug)

\+ **pfunc** Prolog functor (atom)
\+ **tname** table name (atom or LPA string)
\+ **aug** augmentation string (LPA string)

---

db_attach/4

Connects a database table to a Prolog functor

db_attach(HDBC, pfunc, tname, aug)

\+ **HDBC** database connection number (integer)

\+ **pfunc** Prolog functor (atom)
\+ **tname** table name (atom or LPA string)
\+ **aug** augmentation string (LPA string)

---

db_begintran/0

Turns autocommit off

db_begintran

---

db_begintran/1

Turns autocommit off

db_begintran(+HDBC)

\+ **HDBC** database connection number (integer)

---

db_close_cursor/1

Close a cursor for a query

db_close_cursor(Cursor)

\+ **Cursor** cursor for the query (integer)

---

db_commit/0

Commit transaction

db_commit.

---

db_commit/1

Commit transaction

db_commit(HDBC)

\+ **HDBC** database connection number (integer)

---

db_connect/1

To connect to an ODBC data source

db_connect(datasource)

\+ **datasource** data source name (atom/LPA string)

---

db_connect/2

To connect to an ODBC data source

db_connect(datasource, HDBC)

\+ **datasource** data source name (atom/LPA string)

\- **HDBC** database connection number (integer)

---

db_create_index/4

To create an index on a database table

db_create_index(tname, iname, uflag, colspec)

\+ **tname** table name (atom or LPA string)

\+ **iname** index name (atom or LPA string)

\+ **uflag** unique or non_unique (atom)

\+ **colspec** column(s) to form index (atom/LPA string)

---

db_create_index/5

To create an index on a database table

db_create_index(HDBC, tname, iname, uflag, colspec)

\+ **HDBC** database connection number (integer)

\+ **tname** table name (atom or LPA string)

\+ **iname** index name (atom or LPA string)

\+ **uflag** unique or non_unique (atom)

\+ **colspec** column(s) to form index (atom/LPA string)

---

db_create_table/2

To create a database table

db_create_table(tname, colspec)

\+ **tname** table name (atom or LPA string)

\+ **colspec** field descriptions (atom or LPA string)

```
?- db_create_table( 'DEPT', 'DEPTNO NUMBER,DNAME
CHAR(14),LOC CHAR(13)' ).<return>
SQL> CREATE TABLE DEPT(DEPTNO NUMBER,DNAME CHAR(14),LOC
CHAR(13))
yes
```

---

db_create_table/3

To create a database table

db_create_table(HDBC, tname, colspec)

\+ **HDBC** database connection number (integer)

\+ **tname** table name (atom or LPA string)

\+ **colspec** field descriptions (atom or LPA string)

---

db_cursor_tuple/2

Retrieve a tuple for a cursor

db_cursor_tuple(Cursor, R)

\+ **Cursor** Cursor for query (integer)

\- **R** tuple for the cursor (list)

---

db_cursor_tuple/3

Retrieve a tuple for a cursor

db_cursor_tuple(HDBC, Cursor, R)

+ **HDBC** database connection number (integer)

+ **Cursor** Cursor for query (integer)

- **R** tuple for the cursor (list)

---

db_delete_index/1

To delete an index on a database table

db_delete_index(iname)

+ **iname** index name (atom or LPA string)

```
?- db_delete_index( 'EMP_KEY' ).<return>
SQL> DROP INDEX EMP_KEY
yes
```

---

db_delete_index/2

To delete an index on a database table

db_delete_index(HDBC, iname)

+ **HDBC** database connecton number (integer)

+ **iname** index name (atom or LPA string)

---

db_delete_record/2

To delete a record from a database table

db_delete_record(tname, fspec)

\+ **tname** table name (atom or LPA string)

\+ **fspec** fields to identify record (list)

```
?- db_delete_record( 'Shippers', [4|_] ). <return>
SQL> DELETE FROM Shippers WHERE (ShipperID=4)
yes

?- db_delete_record( 'Shippers', [_,'LPA',_] ).<return>
SQL> DELETE FROM Shippers WHERE (CompanyName='LPA')
yes
```

---

db_delete_record/3

To delete a record from a database table

db_delete_record(HDBC, tname, fspec)

\+ **HDBC** database connection number (integer)

\+ **tname** table name (atom or LPA string)

\+ **fspec** fields to identify record (list)

---

db_delete_table/1

To delete a database table

db_delete_table(tname)

\+ **tname** table name (atom or LPA string)

---

db_delete_table/2

To delete a database table

db_delete_table(HDBC, tname)

+ **HDBC** database connection number (integer)

+ **tname** table name (atom or LPA string)

---

db_delete_view/1

To delete a database view

db_delete_view(vname)

+ **vname**database view name (atom or LPA string)

---

db_delete_view/2

To delete a database view

db_delete_view(HDBC, vname)

+ **HDBC** database connection number (integer)

+ **vname**database view name (atom or LPA string)

---

db_disconnect/0

To disconnect from an ODBC data source

db_disconnect

---

db_disconnect/1

To disconnect from an ODBC data source

db_disconnect(HDBC)

+ **HDBC** database connection number (integer)

---

db_flag/3

Set display of error messages and SQL on/off

db_flag(flag, old, new)

\+ **flag** show_sql (atom)

show_db_error (atom)

default_connection (atom)

match null

record_db_error

\- **old** = on or off (atom)

\+ **new** = on or off (atom)

Note: default_connection => old/new = integer

---

db_get_byte_at_address/3

Get the value of a byte at a given address

db_get_byte_at_address(address, offset, val)

\+ **address** $address$(+address) (structure)

\+ **offset** byte offset from address (integer)

\- **val** the byte's value

---

db_get_schema/2

Get various schema information

db_get_schema(flag, res)

\+ **flag** arity(+tname) (structure),
columns(+tname) (structure),

accessible (atom)

user (atom)
sources (atom)

- **res** result list depending on input (list)

---

db_get_schema/3

Get various schema information

db_get_schema(HDBC, flag, res)

\+ **HDBC** database connection number (integer)

\+ **flag** arity(+tname) (structure),
columns(+tname) (structure),

accessible (atom)

user (atom)
sources (atom)

- **res** result list depending on input (list)

Notes: HDBC cannot be specified when flag = sources

---

db_halt/0

Shut all open connections and unload the DLL

db_halt

---

db_open_cursor/2

Open a cursor for a query

db_open_cursor(Query, Cursor)

\+ **Query** SQL for query (atom or LPA string)

- **Cursor** cursor for the query (integer)

---

db_open_cursor/3

Open a cursor for a query

db_open_cursor(HDBC, Query, Cursor)

\+ **HDBC** database connection number (integer)

\+ **Query** SQL for query (atom or LPA string)

\- **Cursor** cursor for the query (integer)

---

db_rollback/0

Roll back (discard) transaction

db_rollback

---

db_rollback/1

Roll back (discard) transaction

db_rollback(HDBC)

\+ **HDBC** - database connection number (integer)

---

db_set_max_blob_size/1

Set the maximum field projection size

db_set_max_blob_size(Size)

\+ **Size** maximum field size (integer)

---

db_show_schema/1

Show various schema information

db_show_schema(flag)

\+ **flag** arity(+tname) (structure),
columns(+tname) (structure),

accessible (atom)
user (atom)
sources (atom)

Output: displays depending on input

---

db_show_schema/2

Show various schema information

db_show_schema(HDBC, flag)

\+ **HDBC** database connection number (integer)

\+ **flag** arity(+tname) (structure),
columns(+tname) (structure),

accessible (atom)
user (atom)
sources (atom)

Output: displays depending on input
Note: HDBC cannot be used with flag = sources

---

db_sql/1

Executes any non-retrieval SQL statement

db_sql(sql)

\+ **sql** SQL 'non-select' command (atom or LPA string)

```
?- db_sql( 'ALTER TABLE DEPT ADD COLUMN TEL CHAR(15)' ).
<return>
SQL> ALTER TABLE DEPT ADD COLUMN TEL CHAR(15)
yes
```

```
?- db_sql( 'DELETE FROM DEPT;' ). <return>
SQL> DELETE FROM DEPT;
yes

?- db_sql( 'DROP TABLE DEPT ;' ). <return>
SQL> DROP TABLE DEPT ;
yes
```

---

db_sql/2

Executes any non-retrieval SQL statement

db_sql(HDBC, sql)

\+ **HDBC** database connection number (integer)

\+ **sql** SQL 'non-select' command (atom or LPA string)

---

db_sql_select/2

Database access request

db_sql_select([+HDBC], +sql, -row)

\+ **sql** SQL query (atom or LPA string)

\- **row** Values from table (list)

```
?- findall( Result, db_sql_select( 'select CompanyName
from Shippers', [Result] ), CompanyNames ). <return>
SQL> select CompanyName from Shippers
Result = _ ,
CompanyNames = ['Speedy Express','United
Package','Federal Shipping','Keylink Computers']
```

---

db_sql_select/3

Database access request

db_sql_select(HDBC, sql, row)

\+ **HDBC** database connection number (integer)

\+ **sql** SQL query (atom or LPA string)

- **row** Values from table (list)

---

db_tuple/2

To retrieve a record from a database table

db_tuple(tname, row)

\+ **tname** tname name (atom or LPA string)
\- **row** the record as a list of fields (list)

---

db_tuple/3

To retrieve a record from a database table

db_tuple(tname, aug, row)

\+ **tname** tname name (atom or LPA string)
\+ **aug** augmentation string (LPA string)

\- **row** the record as a list of fields (list)

---

db_tuple/4

To retrieve a record from a database table

db_tuple(HDBC, tname, aug, row)

\+ **HDBC** database connection number (integer)

\+ **tname** tname name (atom or LPA string)
\+ **aug** augmentation string (LPA string)

\- **row** the record as a list of fields (list)

INDEX

—A—

Accessing a database table, 11

Adding and deleting records, 27

Alternative access to DBMS tables, 16

any SQL query, 17

arithmetic operators, 16

Augmenting queries for attached tables, 12

—B—

backtrack, 16

Backtracking, 11

—C—

Concurrent access, 5

Connecting to a DBMS, 9

COUNT, 18

Creating tables and indexes, 26

cursor manipulation, 5

—D—

data dictionary, 5

DATA DICTIONARY, 21

data dictionary information, 11

Data dictionary tables, 25

Data Source Name, 9

Data sources, 21

Database interface error, 40

Database tables, 21

db_add_record, 27, 42

db_attach, 43

db_begintran, 31

db_begintran(+dbms)., 43

db_commit, 31, 44

db_connect, 9, 44

db_create_index, 26, 44

db_create_table, 26, 45

db_delete_index, 27, 45, 46

db_delete_record, 28, 46

db_delete_table, 26, 46

db_delete_view, 27, 47

db_disconnect, 47, 48

db_flag, 47

db_flag/3, 29

db_get_byte_at_address, 48

db_get_byte_at_address/4, 15

db_get_schema, 48

db_rollback, 31, 49

db_show_schema, 49

db_sql, 28, 50, 51

db_sql_select/3, 17

db_tuple, 16, 51

dblink.pc, 7

DBMS error messages, 29

debugging, 40

default connection, 30

Deleting tables, views and indexes, 26

Disconnecting from a Database, 10

DSN, 9

—E—

Ending a Prolog session, 11

enroling new users, 28

—F—

Failed to open Prodata DLL:lpadbw.dll, 39

floating point, 14

—I—

integer, 14

—J—

Joins, 16

—L—

listing, 11

Long values, 14

lpadbw.dll, 7

LPADBW.DLL, 39

67

LPA/DBMS Prodata interface

—N—

Non-defined fields, 15

Null fields, 15

—O—

ODBC administration program, 7

ODBC Administrator, 40

ODBC driver, 7

—P—

password protected, 40

—R—

Relational Operators, 16

repair, 40

ROLLBACK, 31

—S—

Seeing the SQL for a generated query, 29

server-based DBMS, 9

Specifying connection for db_attach, 13

SQL commands, 28

string operators, 16

—T—

Table arity, 23

Table columns, 23

Table tuples, 24

time, 14

tracing, 40

Transactions, 31

transparency, 5

Type conversions, 14

—U—

unification, 18

unit ground clauses, 5

unquoted atoms, 14

—W—

WHERE, 18