Managing external tables for SWI-Prolog

Jan Wielemaker
Human Computer Studies (HCS),
University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@uva.nl

March 16, 2011

Abstract

This document describes a foreign language extension to SWI-Prolog for the manipulation of 'external tables'. External tables are files using a textual representation of records separated into fields. The package allows for a flexible definition of the format of the file in terms of records and fields, how the information in the file should be mapped onto Prolog data types and what properties the file has to improve the performance of lookup.

The table package has been used successfully to deal with large static databases such as dictionaries. Compared to loading the tables into the Prolog database, this approach required much less memory and loads much faster while providing reasonable lookup-performance on sorted tables.

This package uses read-only 'mapping' of the database file into memory and is ported to Win32 (Windows 95 and NT) as well as Unix systems providing the mmap() system call (Solaris, SunOs, Linux and many more modern Unices).

Contents

1 Introduction

Prolog programs sometimes need access to large sets of background data. For example in the GRASP project we need access to ontologies of art objects, a large lexicon and translation dictionaries. Storage of such information as Prolog clauses is not sufficiently efficient in terms of the memory requirements.

The table package outlined in this document allows for easy access of large structured files. The package uses binary search if possible and linear search for queries that cannot use more efficient algorithms without building additional index tables. Caching is achieved using the file-to-memory maps supported by many modern operating systems.

The following sections define the interface predicates for the package. Section 4 provides an example to access the Unix password file.

2 Managing external tables

2.1 Creating and destroying tables

This section describes the predicates required for creating and destroying the access to external database tables.

new_table(*+File, +Columns, +Options, -Handle***)**

Create a description of a new table, stored in *File*. *Columns* is a list of descriptions for each column. A column description is of the form

ColumnName(*Type [, ColumnOptions]*)

Type denotes the Prolog type to which the field should be converted and is one of:

`integer`	Convert to a Prolog integer. The input is treated as a decimal number.
`hexadecimal`	Convert to a Prolog integer. The input is treated as a hex number.
`float`	Convert to a Prolog floating point number. The input is handled by the C-library function `strtod()`.
`atom`	Convert to a Prolog atom.
`string`	Convert to a SWI-Prolog string object.
`code_list`	Convert to a list of ASCII codes.

ColumnOptions is a list of additional properties of the column. Supported values are:

`sorted`	The field is strictly sorted, but may have (adjacent) duplicate entries. If the field is textual, it should be sorted alphabetically, otherwise it should be sorted numerically.
`sorted(+`*Table*`)`	The (textual) field is sorted using the ordering declared by the named *ordering table*. This option may be used to define reverse order, 'dictionary' order or other irregular alphabetical ordering. See `new_order_table/2`.
`unique`	This column has distinct values for each row in the table.
`downcase`	Map all uppercase in the field to lowercase before converting to a Prolog atom, string or code_list.
`map_space_to_underscore`	Map spaces to underscores before converting to a Prolog atom, string or code_list.
`syntax`	For numerical fields. If the field does not contain a valid number, matching the value fails. Reading the value returns the value as an atom.
`width(+`*Chars*`)`	Field has fixed width of the specified number of characters. The column-separator is not considered for this column.
`arg(+`*Index*`)`	For `read_table_record/4`, unify the field with the given argument of the record term. Further fields will be assigned index+1, ...
`skip`	Don't convert this field to Prolog. The field is simply skipped without checking for consistency.

The *Options* argument is a list of global options for the table. Defined options are:

`record_separator(+Code)`	Character (ASCII) value of the character separating two records. Default is the newline (ASCII 10).
`field_separator(+Code)`	Character (ASCII) value of the character separating two fields in a record. Default is the space (ASCII 32), which also has a special meaning. Two fields separated by a space may be separated by any non-empty sequence of spaces and tab (ASCII 9) characters. For all other separators, a single character separates the fields.
`escape(+Code, +ListOfMap)`	Sometimes, a table defines escape sequences to make it possible to use the separator-characters in text-fields. This options provides a simple way to handle some standard cases. *Code* is the ASCII code of the character that leads the escape sequence. The default is `-1`, and thus never matched. *ListOfMap* is a list of *From* `=` *To* character mappings. The default map table is the identity map, unless *Code* refers to the `\` character, in which case `\b`, `\e`, `\n`, `\r` and `\t` have their usual meaning.
`functor(+Head)`	Functor used by `read_table_record/4`. Default is `record` using the maximal argument index of the fields as arity.

If the options are parsed successfully, *Handle* is unified with a term that may be used as a handle to the table for future operations on it. Note that `new_table/4` does not access the file system, so its success only indicates the description could be parsed, not the presence, access or format of the file.

open_table(*+Handle***)**

Open the table. This predicate normally does not need to be called explicitely, as all operations on the table handle will automatically open the table if this is required. It fails if the file cannot be accessed or some other error with the required operating-system resources occurs. The contents of the file is not examined by this predicate.

close_table(*+Handle***)**

Close the file and other system resources, but do not remove the description of the table, so it can be re-opened later.

free_table(*+Handle***)**

Close and remove the handle. After this operation, *Handle* becomes invalid and further references to it causes undefined behaviour.

2.2 Accessing a table

This section describes the predicates to read data from a table.

2.2.1 Finding record locations in a table

Records are addressed by their offset in the table (file). As records have generally non-fixed length, searching is often required. The predicates below allow for finding records in the file.

get_table_attribute(*+Handle, +Attribute, -Value***)**

Fetch attributes of the table. Defined attributes:

`file` Unify value with the name of the file with which the table is associated.

`field(`*N*`)` Unify value with declaration of n-th (1-based) field.

`field_separator` Unify value with the field separator character.

`record_separator` Unify value with the record separator character.

`key_field` Unify value with the 1-based index of the field that is sorted or fails if the table contains no sorted fields.

`field_count` Unify value with the total number of columns in the table.

`size` Unify value with the number of characters in the table-file, **not** the number of records.

`window` Unify value with a term *Start* `-` *Size*, indicating the properties of the current window.

table_window(*+Handle, +Start, +Size***)**

If only part of the file represents the table, this call may be used to define a window on the file. *Start* defines the start of the window relative to the start of the file. *Size* is the size in characters. Skipping a header is one of the possible purposes for this call.

table_start_of_record(*+Handle, +From, +To, -Start***)**

Enumerates (on backtracking) the start of records in the table in the region [From, To). Together with `read_table_record/4`, this may be used to read the table's data.

table_previous_record(*+Handle, +Here, -Previous***)**

If *Here* is the start of a record, find the start of the record before it. If *Here* points at an arbitrary location in a record, the start of this record will be returned.

2.2.2 Reading records

There are two predicates for reading records. The `read_table_record/4` reads an entire record, while `read_table_fields/4` reads one or more fields from a record.

read_table_record(*+Handle, +Start, -Next, -Record***)**

Read a record from the table. *Handle* is a handle as returned by `new_table/4`. *Start* is the location of a record. If *Start* does not point to the start of a record, this predicate searches backwards for the starting position. *Record* is unified with a term constructed from the *functor* associated with the table (default name `record` and arity the number of not-skipped columns), each of the arguments containing the converted data. An error is raised if the data could not be converted. *Next* is unified with the start position for the next record.

read_table_fields(*+Handle, +Start, -Next, -Fields***)**

As `read_table_record/4`, but *Fields* is a list of terms *+Name*(-*Value*), and the *Values* will be unified with the values of the specified field.

read_table_record_data(*+Handle, +Start, -Next, -Record***)**

Similar to `read_table_record/4`, but unifies record with a Prolog string containing the data of the record unparsed. The returned record does **not** contain the terminating record-separator.

2.2.3 Searching the table

in_table(*+Handle, ?Fields, -RecordPos***)**

Searches the table for records matching *Fields*. If a match is found, the variable (see below) fields in *Fields* are unified with the corresponding field value, and *RecordPos* is unified with the position of the record. The latter handle may be used in a subsequent call to `read_table_record/4` or `read_table_fields/4`.

Fields is a list of field specifiers. Each specifier is of the format:

FieldName(*Value* [, *Options*])

Options is a list of options to specify the search. By default, the package will search for an exact match, possibly using the ordering table associated with the field (see `order` option in `new_table/4`). Options are:

`prefix`	Uses prefix search with the default table.
`prefix(`*Table*`)`	Uses prefix search with the specified ordering table.
`substring`	Searches for a substring in the field. This requires linear search of the table.
`substring(`*Table*`)`	Searches for a substring, using the table information for determining the equivalence of characters.
`=`	Default equivalence.
`=(`*Table*`)`	Equivalence using the given table.

If *Value* is unbound (i.e. a variable), the record is considered not specified. The possible option list is ignored. If a match is found on the remaining fields, the variable is unified with the value found in the field.

First, the system checks whether there is an ordered field that is specified. In this case, binary search is employed to find the matching record(s). Otherwise, linear search is used.

If the match contains a specified field that has the property `unique` set (see `new_table/4`), `in_table/3` succeeds deterministically. Otherwise it will create a backtrack-point and backtracking will yield further solutions to the query.

`in_table/3` may be comfortable used to bind the table transparently to a predicate. For example, we have a file with lines of the format.[1]

[1] This is the `disproot.dat` table from the AAT database used in GRASP

```
C1C2,Full Name
```

C1C2 is a two-character identifier used in the other tables, and *FullName* is the description of the identifier. We want to have a predicate identifier_name(?Id, ?FullName) to reflect this table. The code below does the trick:

```
:- dynamic stored_idtable_handle/1.


idtable(Handle) :-
        stored_idtable_handle(Handle).
idtable(Handle) :-
        new_table('disproot.dat',
                  [ id(atom, [downcase, sorted, unique]),
                    name(atom)
                  ],
                  [ field_separator(0',)
                  ], Handle),
        assert(stored_idtable_handle(Handle)).

identifier_name(Id, Name) :-
        idtable(Handle),
        in_table(Handle, [id(Id), name(Name)], _).
```

2.2.4 Miscellaneous

table_version(*-Version, -CompileDate*)
Unify *Version* with an atom identifying the version of this package, and *CompileDate* with the date this package was compiled.

3 Flexible ordering and equivalence based on character table

This package was developed as part of the GRASP project, where it is used for browsing lexical and ontology information, which is normally stored using 'dictionary' order, rather than the more conventional alphabetical ordering based on character codes. To achieve programmable ordering, the table package defines 'order tables'. An order table is a table with the cardinality of the size of the character set (256 for extended ASCII), and maps each character onto its 'order number', and some characters onto special codes.

The default (`exact`) table matches all character codes onto themselves. The default `case_insensitive` table matches all uppercase characters onto their corresponding lowercase character. The tables `iso_latin_1` and `iso_latin_1_case_insensitive` map the ISO-latin-1 letters with diacritics into their plain counterpart.

To support dictionary ordering, the following special categories are defined:

ignore	Characters of the ignore set are simple discarded from the input.
break	Characters from the break set are treated as word-breaks, and each non-empty sequence of them is considered equal. A word break precedes a normal character.
tag	Characters of type tag indicate the start of a 'tag' that should not be considered in ordering, unless both strings are the same upto the tag.

The following predicates are defined to manage and use these tables:

new_order_table(*+Name, +Options***)**

Create a new, or replace the order-table with the given name (an atom). *Options* is a list of options:

`case_insensitive`	Map all upper- to lowercase characters.
`iso_latin_1`	Start with an ISO-Latin-1 table
`iso_latin_1_case_insensitive`	Start with a case-insensitive ISO-Latin-1 table
`copy(+`*Table*`)`	Copy all entries from *Table*.
`tag(+`*ListOfCodes*`)`	Add these characters to the set of 'tag' characters.
`ignore(+`*ListOfCodes*`)`	Add these characters to the set of 'ignore' characters.
`break(+`*ListOfCodes*`)`	Add these characters to the set of 'break' characters.
`+`*Code1* `= +`*Code2*	Map *Code1* onto *Code2*.

order_table_mapping(*+Table, ?From, ?To***)**

Read the current mapping. *To* is a character code or one of the atoms `break`, `ignore` or `tag`.

compare_strings(*+Table, +S1, +S2, -Result***)**

Compare two strings using the named *Table*. *S1* and *S2* may be atoms, strings or code-lists. *Result* is one of the atoms `<`, `=` or `>`.

prefix_string(*+Table, +Prefix, +String***)**

Succeeds if *Prefix* is a prefix of *String* using the named *Table*.

prefix_string(*+Table, +Prefix, -Rest, +String***)**

Succeeds if *Prefix* is a prefix of *String* using the named *Table*, and *Rest* is unified with the remainder of *String* that is not matched. Please note that the existence of an order-table implies simple contatenation using `atom_concat/3` cannot be used to determine the non-matched part of the string.

sub_string(*+Table, +Sub, +String***)**

Succeeds if *Sub* is a substring of *String* using the named *Table*.

4 Example: accessing the Unix passwd file

The Unix passwd file is a file with records spanning a single line each. The fields are separated by a single ‘:’ character. Here is an example of a line:

```
joe:hgdu3r3bce:53:100:Joe Johnson:/users/joe:/bin/bash
```

The following call defines a table for it:

```
?- new_table(’/etc/passwd’,
             [ user(atom),
               passwd(code_list),
               uid(integer),
               gid(integer),
               gecos(code_list),
               homedir(atom),
               shell(atom)
             ],
             [ field_separator(0’:)
             ],
             H).
```

To find all people of group *100*, use:

```
?- findall(User, in_table(H, [user(User), gid(100)], _), Users).
```