1

DBD::CSV and DBD::File

Version

Version 0.1019.

`DBD::File` is an abstract general purpose driver for treating files as database tables and designed to be subclassed rather than used directly. `DBD::CSV` is a subclass of `DBD::File` for working with files that typically store one row per text line with fields separated with a comma, semicolon or tab character, typically called CSV files.

Author and Contact Details

The driver author is Jochen Wiedmann. He can be contacted via the *dbi-users* mailing list.

Supported Database Versions and Options

The `DBD::File` driver works with the `SQL::Statement` module, version 0.1011 or later. This module is a simple SQL parser and evaluator. In particular it is restricted to single table queries. Table joins are not supported.

The `DBD::CSV` driver internally uses the `Text::CSV_XS` module, version 0.16 or later, for reading and writing CSV files.

It's important to note that while just about everyone thinks they know what the CSV file format is, there is actually no formal definition of the format and there are many subtle differences.

Connect Syntax

The DBI->connect() Data Source Name, or *DSN*, can be one of the following:

```
dbi:CSV:
dbi:CSV:attrs
```

where attrs is an optional semicolon-separated list of *key=value* pairs. Note that you must not use dbi:File:, as the DBD::File driver is an abstract superclass and not usable by itself.

Known attributes include:

f_dir=ltdirectorygt

By default files in the current directory are treated as tables. The attribute *f_dir* makes the module open files in the given directory.

csv_eol

csv_sep_char

csv_quote_char

csv_escape_char

These attributes are used for describing the CSV file format in use. For example, to open */etc/passwd*, which is colon-separated and line-feed terminated, as a table, one would use:

```
csv_eol=\n;csv_sep_char=:
```

The defaults are \r\n, comma (,), double-quote ("), and double-quote (") respectively. All of these attributes and defaults are inherited from the Text::CSV_XS module.

Numeric Data Handling

Without question, the main disadvantage of the DBD::CSV module is the lack of appropriate type handling. While reading a CSV table you have no way to reliably determine the correct data type of the fields. All fields look like strings and are treated as such by default.

The SQL::Statement module, and hence the DBD::CSV driver, accepts the numeric types INTEGER and REAL in CREATE TABLE statements, but they are always stored as strings and, by default, retrieved as strings.

It is possible to read individual columns as integers or doubles, in which case they are converted to Perl's internal data types IV and NV, integer and numeric value respectively. Unsigned values are not supported.

To assign certain data types to columns, you have to create *metadata definitions*. The following example reads a table *table_name* with columns *I*, *N*, and *P* of type integer, double, and string, respectively:

```
my $dbh = DBI->connect("DBI:CSV:");
$dbh->{csv_tables}->{table_name}->{types} =
    [ C<Text::CSV_XS>::IV(), C<Text::CSV_XS>::NV(),
      C<Text::CSV_XS>::PV() ];
# Note, we assume a certain order of I, N and P!
my $sth = $dbh->prepare("SELECT * FROM foo");
```

String Data Handling

Similar to numeric values, `DBD::CSV` accepts more data types in `CREATE TABLE` statements than it really supports. You can use `CHAR(`*n*`)` and `VARCHAR(`*n*`)` with arbitrary numbers *n*, BLOB, or TEXT, but in fact these are always BLOBs, in a loose kind of way.

The one underlying string type can store any binary data including embedded NUL characters. However, many other CSV tools may choke if given such data.

Date Data Handling

No date or time types are directly supported.

LONG/BLOB Data Handling

BLOBs are equivalent to strings. They are only limited in size by available memory.

Other Data Handling issues

The `type_info_all()` method is supported and returns the types VARCHAR, CHAR, INTEGER, REAL, BLOB and TEXT.

Transactions, Isolation and Locking

The driver doesn't support transactions.

No explicit locks are supported. Tables are locked while statements are executed, but the lock is immediately released once the statement is completed.

No-Table Expression Select Syntax

You can only retrieve table data. It is not possible to select constants.

Table Join Syntax

Table joins are not supported.

Table and Column Names

Table and column names are case sensitive. However, you should consider that table names are in fact file names, so tables *Foo* and *foo* may both be present with the same data. However, they may be subject to different metadata definitions in `$dbh->{'csv_tables'}`.

See for more details on table and column names.

Case Sensitivity of LIKE Operator

Two different LIKE operators are supported. LIKE is case sensitive, whereas CLIKE is not.

Row ID

Row IDs are not supported.

Automatic Key or Sequence Generation

Neither automatic keys nor sequences are supported.

Automatic Row Numbering and Row Count Limiting

Neither automatic row numbering nor row count limitations are supported.

Parameter Binding

Question marks are supported as placeholders, as in:

```
$dbh->do("INSERT INTO A VALUES (?, ?)", undef, $id, $name);
```

The `:1` placeholder style is not supported.

Stored Procedures

Stored procedures are not supported.

Table Metadata

By default the driver expects the column names being stored in the tables first row, as in:

```
login:password:uid:gid:comment:shell:homedir
root:s34hj34n34jh:0:0:Superuser:/bin/bash:/root
```

If column names are not present, you may specifiy column names via:

```
$dbh->{csv_tables}->{$table}->{skip_rows} = 0;
$dbh->{csv_tables}->{$table}->{col_names} =
    [qw(login password uid gid comment shell homedir)];
```

in which case the first row is handled as a data row.

If column names are not supplied and not read from the first row, the names *col0*, *col1*, ... are generated automatically. Column names can be retrieved via the standard `$sth->{NAME}` attribute.

The *NULLABLE* attribute returns an array of all ones. Other metadata attributes are not supported.

The table names, or file names, can be read via `$dbh->table_info()` or `$dbh->tables()` as usual.

Driver-specific Attributes and Methods

Besides the attributes *f_dir*, *csv_eol*, *csv_sep_char*, *csv_quote_char* and *csv_sep_char* that have already been discussed above, the most important database handle attribute is:

```
$dbh->{csv_tables}
```

`csv_tables` is used for specifying table metadata. It is an hash ref with table names as keys, the values being hash refs with the following attributes:

file

The file name being associated to the table. By default, the file name is `$dbh->{f_dir}/$table`.

col_names

An array ref of column names.

skip_rows

This number of rows will be read from the top of the file before reading the table data, and the first of those will be treated as an array of column names. However, the *col_names* attribute takes precedence.

types

This is an array ref of the `Text::CSV_XS` type values for the corresponding columns. Three types are supported and their values are defined by the `IV()`, `NV()`, and `PV()` functions in the `Text::CSV_XS` package.

There are no driver specific statement handle attributes and no private methods for either type of handle.

Positioned updates and deletes

Positioned updates and deletes are not supported.

Differences from the DBI Specification

The statement handle attributes *PRECISION*, *SCALE*, and *TYPE* are not supported.

Also note that many statement attributes cease to be available after fetching all the result rows or calling the `finish()` method.

URLs to More Database/Driver Specific Information

```
http://www.whatis.com/csvfile.htm
```

Concurrent use of Multiple Handles

The number of database handles is limited by memory only. It is recommended to use multiple database handles for different table formats.

There are no limitations on the use of multiple statement handles from the same `$dbh`.

The driver is believed to be completely thread safe.