NAME

DBI::SQL::Nano - a very tiny SQL engine

SYNOPSIS

```
BEGIN { $ENV{DBI_SQL_NANO}=1 } # forces use of Nano rather than SQL::Statement
use DBI::SQL::Nano;
use Data::Dumper;
my $stmt = DBI::SQL::Nano::Statement->new(
"SELECT bar,baz FROM foo WHERE qux = 1"
) or die "Couldn't parse";
print Dumper $stmt;
```

DESCRIPTION

`DBI::SQL::Nano` is meant as a *very* minimal SQL engine for use in situations where SQL::Statement is not available. In most situations you are better off installing SQL::Statement although DBI::SQL::Nano may be faster for some **very** simple tasks.

DBI::SQL::Nano, like SQL::Statement is primarily intended to provide a SQL engine for use with some pure perl DBDs including DBD::DBM DBD::CSV DBD::AnyData, and DBD::Excel. It is not of much use in and of itself. You can dump out the structure of a parsed SQL statement, but that is about it.

USAGE

Setting the DBI_SQL_NANO flag

By default, when a DBD uses `DBI::SQL::Nano` the module will look to see if `SQL::Statement` is installed. If it is, SQL::Statement objects are used. If SQL::Statement is not available, DBI::SQL::Nano objects are used.

In some cases, you may wish to use DBI::SQL::Nano objects even if SQL::Statement is available. To force usage of DBI::SQL::Nano objects regardless of the availability of SQL::Statement, set the environment variable DBI_SQL_NANO to 1.

You can set the environment variable in your shell prior to running your script (with SET or EXPORT or whatever), or else you can set it in your script by putting this at the top of the script:

```
BEGIN { $ENV{DBI_SQL_NANO} = 1 }
```

Supported SQL syntax

```
Here's a pseudo-BNF. Square brackets [] indicate optional items;
Angle brackets <> indicate items defined elsewhere in the BNF.

statement ::=
DROP TABLE [IF EXISTS] <table_name>
| CREATE TABLE <table_name> <col_def_list>
| INSERT INTO <table_name> [<insert_col_list>] VALUES <val_list>
| DELETE FROM <table_name> [<where_clause>]
| UPDATE <table_name> SET <set_clause> <where_clause>
| SELECT <select_col_list> FROM <table_name> [<where_clause>]
[<order_clause>]

the optional IF EXISTS clause ::=
* similar to MySQL - prevents errors when trying to drop
a table that doesn't exist

identifiers ::=
* table and column names should be valid SQL identifiers
* especially avoid using spaces and commas in identifiers
* note: there is no error checking for invalid names, some
will be accepted, others will cause parse failures
```

```
table_name ::=
* only one table (no multiple table operations)
* see identifier for valid table names

col_def_list ::=
* a parens delimited, comma-separated list of column names
* see identifier for valid column names
* column types and column constraints may be included but are ignored
e.g. these are all the same:
(id,phrase)
(id INT, phrase VARCHAR(40)
(id INT PRIMARY KEY, phrase VARCHAR(40) NOT NULL)
* you are *strongly* advised to put in column types even though
they are ignored ... it increases portability

insert_col_list ::=
* a parens delimited, comma-separated list of column names
* as in standard SQL, this is optional

select_col_list ::=
* a comma-separated list of column names
* or an asterisk denoting all columns

val_list ::=
* a parens delimited, comma-separated list of values which can be:
* placeholders (an unquoted question mark)
* numbers (unquoted numbers)
* column names (unquoted strings)
* nulls (unquoted word NULL)
* strings (delimited with single quote marks);
* note: leading and trailing percent mark (%) and underscore (_)
can be used as wildcards in quoted strings for use with
the LIKE and CLIKE operators
* note: escaped single quotation marks within strings are not
supported, neither are embedded commas, use placeholders instead

set_clause ::=
* a comma-separated list of column = value pairs
* see val_list for acceptable value formats

where_clause ::=
* a single "column/value <op> column/value" predicate, optionally
preceded by "NOT"
* note: multiple predicates combined with ORs or ANDs are not supported
* see val_list for acceptable value formats
* op may be one of:
< > >= <= = <> LIKE CLIKE IS
* CLIKE is a case insensitive LIKE

order_clause ::= column_name [ASC|DESC]
* a single column optional ORDER BY clause is supported
* as in standard SQL, if neither ASC (ascending) nor
DESC (descending) is specified, ASC becomes the default
```

TABLES

DBI::SQL::Nano::Statement operates on exactly one table. This table will be opened by inherit from DBI::SQL::Nano::Statement and implements the `open_table` method.

```
sub open_table ($$$$$)
{
...
return Your::Table->new( \%attributes );
}
```

DBI::SQL::Nano::Statement_ expects a rudimentary interface is implemented by the table object, as well as SQL::Statement expects.

```
package Your::Table;

use vars qw(@ISA);
@ISA = qw(DBI::SQL::Nano::Table);

sub drop ($$) { ... }
sub fetch_row ($$$) { ... }
sub push_row ($$$) { ... }
sub push_names ($$$) { ... }
sub truncate ($$) { ... }
sub seek ($$$$) { ... }
```

The base class interfaces are provided by DBI::SQL::Nano::Table_ in case of relying on DBI::SQL::Nano or SQL::Eval::Table (see SQL::Eval for details) otherwise.

BUGS AND LIMITATIONS

There are no known bugs in DBI::SQL::Nano::Statement. If you find a one and want to report, please see DBI for how to report bugs.

DBI::SQL::Nano::Statement is designed to provide a minimal subset for executing SQL statements.

The most important limitation might be the restriction on one table per statement. This implies, that no JOINs are supported and there cannot be any foreign key relation between tables.

The where clause evaluation of DBI::SQL::Nano::Statement is very slow (SQL::Statement uses a precompiled evaluation).

INSERT can handle only one row per statement. To insert multiple rows, use placeholders as explained in DBI.

The DBI::SQL::Nano parser is very limited and does not support any additional syntax such as brackets, comments, functions, aggregations etc.

In contrast to SQL::Statement, temporary tables are not supported.

ACKNOWLEDGEMENTS

Tim Bunce provided the original idea for this module, helped me out of the tangled trap of namespaces, and provided help and advice all along the way. Although I wrote it from the ground up, it is based on Jochen Wiedmann's original design of SQL::Statement, so much of the credit for the API goes to him.

AUTHOR AND COPYRIGHT

This module is originally written by Jeff Zucker < jzucker AT cpan.org >

This module is currently maintained by Jens Rehsack < jrehsack AT cpan.org >