A Powerful Sidekick: Using MySQL for High-Volume Data Manipulation in Matlab

by Dimitri Shvorob[1]

1. Introduction.

A continuing poll on WRDS Forum asks visitors to identify the statistical software they most commonly use. SAS and Matlab take top spots in the league table, but SAS's edge is overwhelming: 73% vs. 9%. Broad selection of ready-to-use statistical routines, comprising SAS/STAT, surely plays a big part in explaining the software's appeal. However, one thinks that for many researchers, choice of SAS stems from its superior facility with tasks ancillary to analysis, namely data retrieval and manipulation (subsetting, sorting, reshaping, etc.). Accomplishing these tasks in Matlab is much less convenient.

- Matlab cannot handle large datasets, routinely processed in SAS
- Matlab has no remote-access capabilities similar to those of SAS/CONNECT
- Matlab has no adequate analog to SAS's SQL procedure

Unlike SAS, which leverages available memory resources with continual disk read/writes, Matlab relies on memory exclusively and cannot create, load or save any volume of data exceeding its limits. Although this might not be a problem in most areas of Matlab's application, for WRDS users who routinely manage datasets with hundreds of thousands, or millions, of records, out-of-memory errors are an all-too-familiar occurrence, and switching to SAS the all-too-natural recourse[2].

Regarding the second claim, it suffices to point out that a SAS user can define a directory on WRDS server as a remote library[3], and access SAS datasets located in the directory in the same way as he/she would access a dataset on one's own PC. With Matlab, accessing a MAT file on a different computer is an arduous, if not impossible, task.

Finally, WRDS users frequently perform two types of tasks: (1) match-merging records located in the same or different datasets, and (2) computing summary statistics for groups of observations. Both tasks are easily accomplished in SAS using PROC SQL[4], but require non-trivial programming effort in Matlab, and often produce looped and relatively slow code.

This brief report encourages Matlab users to explore a technique that goes a long way towards resolving the above-mentioned problems[5]. At the core of the proposed approach is use of MySQL Server database management system as (1) a high-capacity data repository accessible to Matlab, and SAS, and (2) a full-fledged SQL processor that can be controlled from Matlab. In essence, we recommend the following course of action when working with WRDS data.

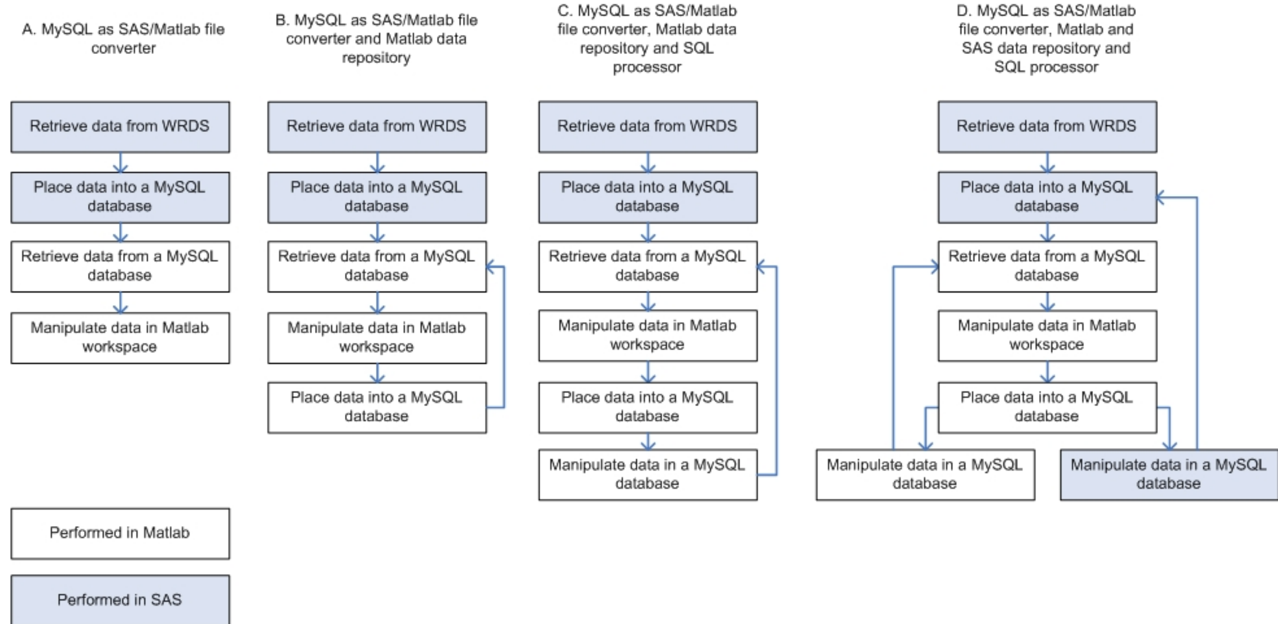
- Retrieve data from WRDS using SAS
- Transfer data to MySQL, converting the SAS dataset into a MySQL table
- Manipulate data within MySQL, submitting SQL commands from Matlab
- Retrieve selected data to Matlab workspace
- If needed, save data in Matlab workspace to a MySQL database

While SAS continues to be needed, its role is limited to getting data from WRDS and passing them to MySQL - all in a single step - after which one can work solely with Matlab. Matlab's memory constraint is not eliminated, but

---

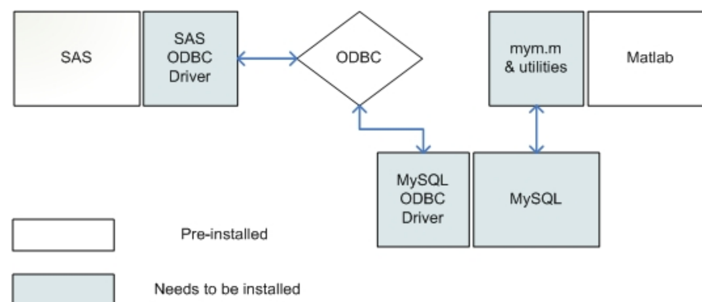
[1]This is the first draft of this report, completed in August 2006, during an internship at WRDS. (The idea of using MySQL in tandem with Matlab was suggested to the author by Michael Boldin). I would like to claim responsibility for any errors, and welcome your comments at dimitri.shvorob@vanderbilt.edu.
[2]However, consult this insightful Mathworks presentation for ways to expand memory resources available to Matlab.
[3]See section 'PC SAS/Connect - Remote Library Services' of this guide for more details.
[4]Summary statistics can also be calculated with PROC MEANS, of course.
[5]The utility functions discussed in section 3 were designed and tested with MySQL 5.0, mym 1.0.8, and Matlab 7. Matlab 6 users may be able to access MySQL through the basic, limited interface of `mym.m`.

with data storage and large-scale data manipulation 'outsourced' to MySQL, the likelihood of it binding is sharply reduced[6].



2. Setup.

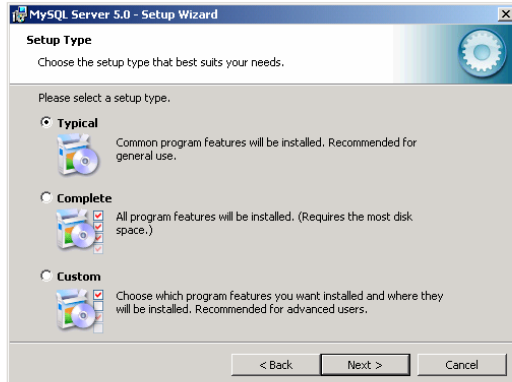
Installing MySQL and the software linking it to Matlab and SA might seem like a challenging task, but requisite setup is, in fact, straightforward. All of the programs are easily downloadable, come with convenient installer modules, and require little or no configuration.

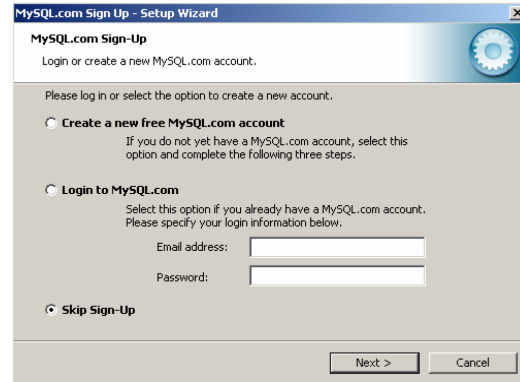


[6]Inquisitive readers may wonder if one could have Matlab communicate with SAS directly, bypassing MySQL's 'middleman'. Indeed, a direct link could be established using functions of Matlab's Database Toolbox, or by operating SAS as a 'COM object' controlled by Matlab. The latter route - illustrated by this submission to Matlab File Exchange - is neither robust, nor easy to follow. Database Toolbox, on the other hand, is an 'add-on' product that needs to be purchased in addition to Matlab, whereas the approach we propose employs free software available to all WRDS users. Even those with access to Database Toolbox will, in our expectation, find the MySQL-based alternative a useful complementary approach.

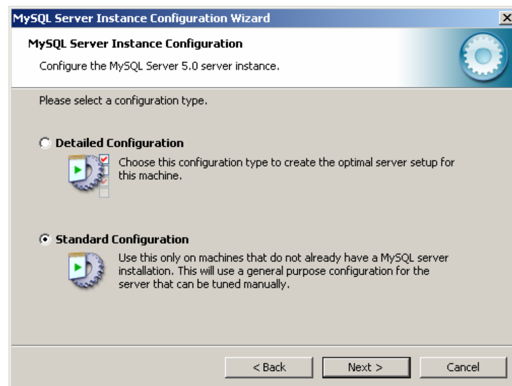
2.1. Installing MySQL.

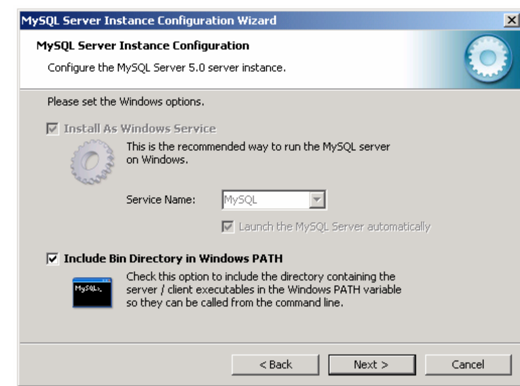
Download and run the installer module of MySQL Server 5.0 (Windows Essentials package), selecting ‘Typical Install’ in ‘Setup type’ screen, skipping sign-up in ‘MySQL.com Sign Up’ screen,



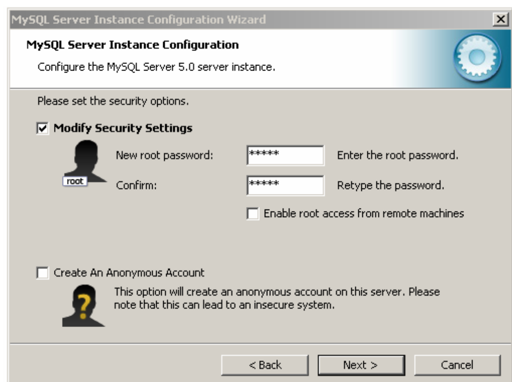


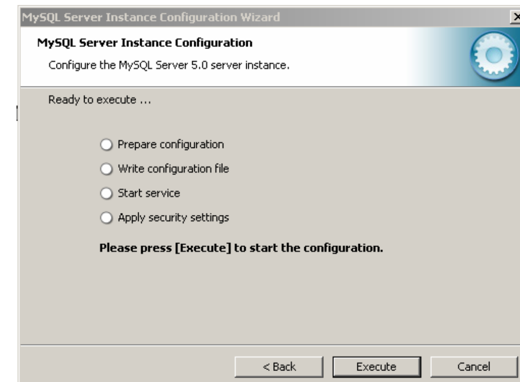
and marking checkbox ‘Configure the MySQL Server now’ in ‘Wizard completed’ screen. Accept default choices in ‘Configuration type’ and ‘Windows options’ screens, and select a password, protecting





access to MySQL databases, in ‘Security options’ screen. (Write the password down, as it will be needed each time you access MySQL, whether from Matlab or SAS). Complete installation of MySQL by pressing ‘Execute’ button in ‘Execute configuration’ screen.



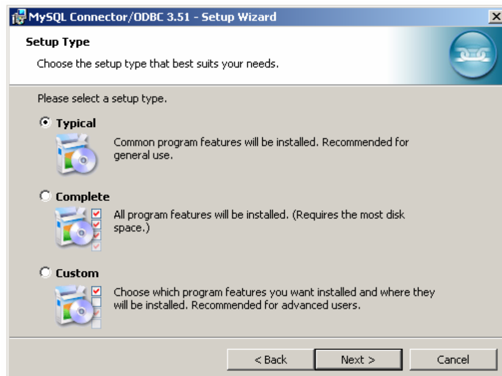


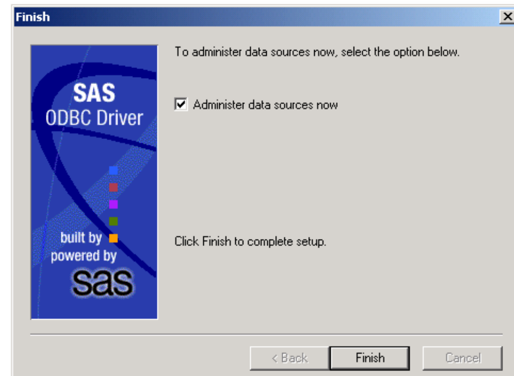
2.2. Connecting MySQL and SAS.

ODBC, or 'open database connectivity', is a Windows/Unix technology allowing data exchange between a wide range of data management systems, including MySQL and SAS. Since neither software package comes with ODBC capability pre-set, one needs to install ODBC 'plug-ins' for MySQL and SAS, and configure 'ODBC data sources' associated with each application, so that the two can be recognized and linked by Windows.

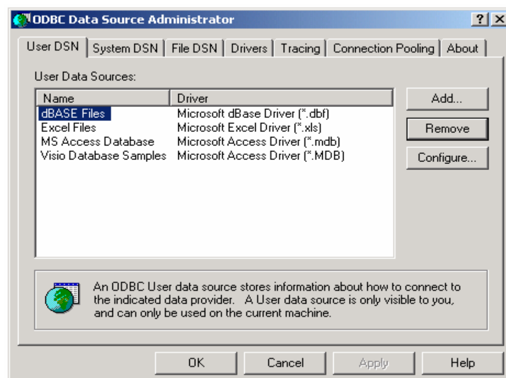
Download and run the installer module of MySQL ODBC driver, selecting 'Typical Install' in 'Setup Type' screen.

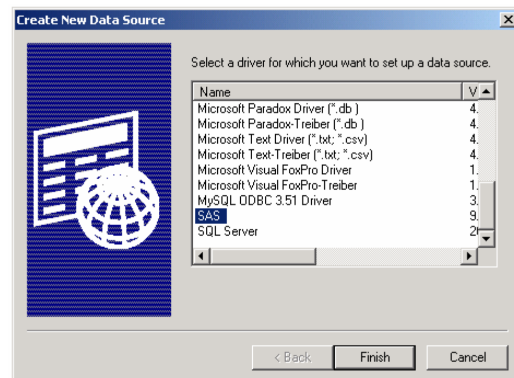
Download and run the installer module of SAS ODBC driver, accepting default settings, and mark 'Administer data sources now' checkbox in 'Finish' screen.





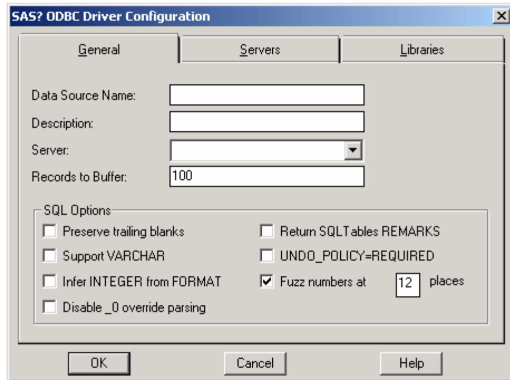
Press 'Finish' to have the installer open 'ODBC Data Source Administrator' system window[7]. Tab 'User DSN' is active, and displays registered ODBC data sources. To add a SAS data source, press 'Add', select SAS from the list of available data sources, and click 'Finish'.

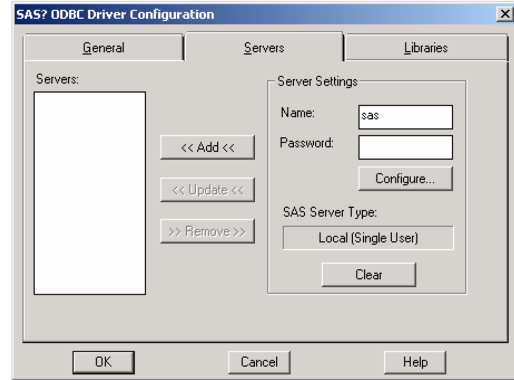




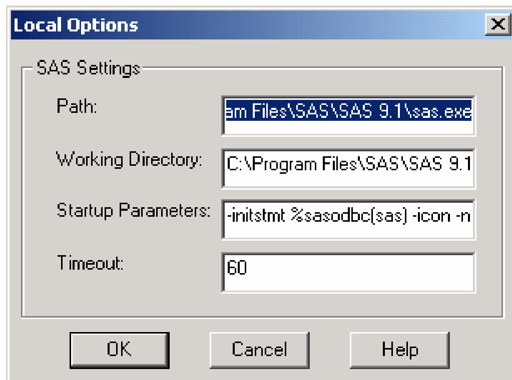
Back in the window of SAS ODBC driver installer, with 'General' tab active, switch to 'Servers' tab, enter an arbitrary name in field 'Name' of 'Server settings' panel, and press 'Configure'.

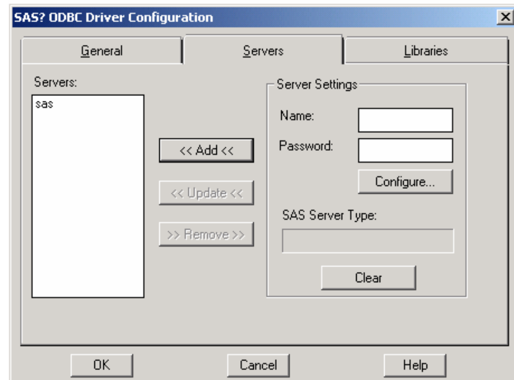
[7]The window can be accessed through Windows Control panel, by navigating to 'Administrative tools' section and clicking on 'Data Sources (ODBC)' icon.



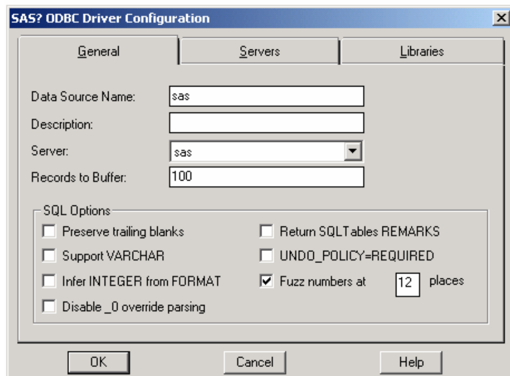


Press ‘OK’ in ‘Local Options’ screen to return to ‘Servers’ tab, then click ‘Add’.





Back in ‘General’ tab, enter an arbitrary name in field ‘Data source name’ and press ‘OK’.

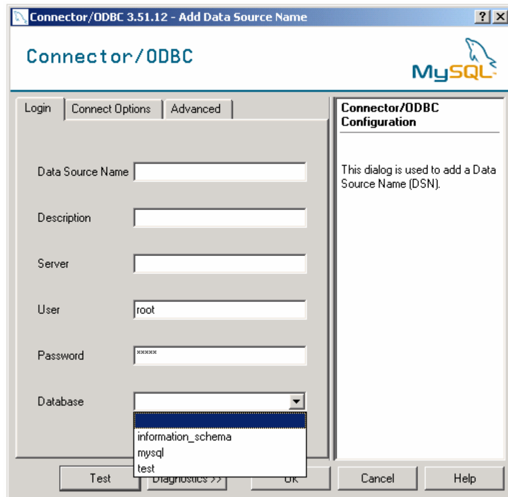


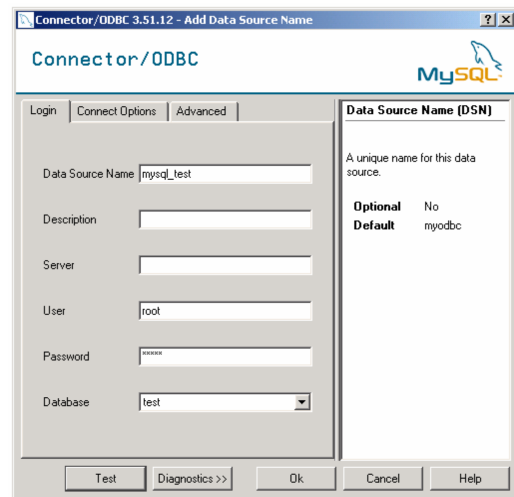
The SAS data source, under the assigned name, can now be seen in the list of registered ODBC data sources of ‘ODBC Data Source Administrator’ window, to which we returned.

A single SAS data source is sufficient for our purposes. For MySQL, on the other hand, a dedicated data source needs to be established for each database that we wish to access with SAS. A brand-new installation of MySQL contains three databases, two of which (`mysql` and `information_schema`) contain system information and are not intended for data storage. Third database, `test`, is an empty ‘starter’ database, which we will set up as a MySQL ODBC data source.

Repeating the initial steps of registering a SAS data source, press 'Add' button in 'User DSN' tab, select 'MySQL ODBC 3.51 Driver' from the list, and press 'Finish'. When MySQL Connector/ODBC configuration screen appears, with 'Login' tab active,

- Enter 'root' in field 'User', and the previously chosen password in field 'Password'
- Select 'test' from the drop-down list in field 'Database'
- Assign an arbitrary name to the MySQL data source, by entering it in field 'Data Source Name'





(Since later you may want to set up and make accessible to SAS additional databases - consider having a database with Compustat data, another with CRSP data, etc. - it is expedient to include the name of the target database into the data source name, to avoid confusion in the future. If you plan to access MySQL databases located on a different computer, e.g. a department or university server, you might also want to distinguish them from those residing on your personal computer, for instance by adding a 'local' or 'remote' keyword to a data source name).

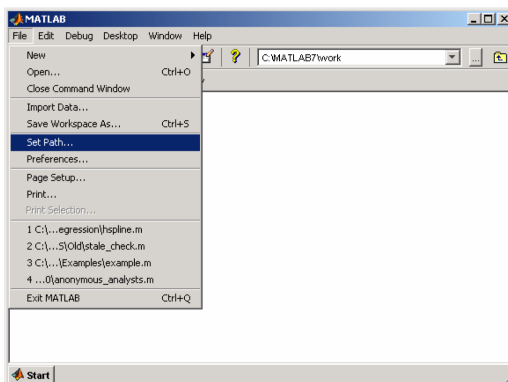
2.3. Connecting MySQL and Matlab.

Matlab functions enabling read/write access to MySQL databases constitute the last, front-end component of the proposed scheme. These include `mym.m`, Yannick Maret's extension of Robert Almgren's `mysql.m`, and a set of utilities based on `mym.m`, written by the author.

Download and run `mym.m` installer, renaming file `mym.mexw32` to `mym.dll` for a release of Matlab 7 older than 7.1.

Download and open the archive containing `mym.m` utilities, listed in Table 1.

Add locations of downloaded m-files to Matlab's path, as shown below.



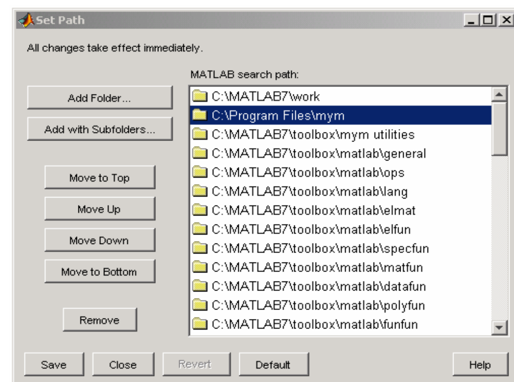


Table 1. Accessing MySQL from Matlab: available functions.

Function	Purpose	Example
`mycheck`	Check MySQL connection	`mycheck`
`myopen`	Connect to MySQL	`myopen('localhost','root','apple')` `myopen('wrds.wharton.upenn.edu','jsmith', 'pear')`
`myclose`	Disconnect from MySQL	`myclose`
`dblist`	List available databases	`all_dbs = dblist`
`dbcurr`	Show current database	`curr_db = dbcurr`
`dbadd`	Create a database	`dbadd('crsp')` `dbadd('project1')`
`dbopen`	Open a database	`dbopen('project1')`
`dbdrop`	Delete a database[8]	`dbdrop('junkdb')`
`tblist`	List database tables	`tblist('project1')`
`tbadd`	Create a table	`tbadd('mytest',{'name','dob','age'},{'varchar(30)','date', 'double'})`
`tbdrop`	Delete a table	`tbdrop('junktb')`
`tbrename`	Rename a table	`tbrename('mytest','test')`
`tbattr`	List column names and types	`[names, types] = tbattr('test')` `names = tbattr('crsp.dsf')`
`tbsize`	Show table's size	`[rows,cols] = tbsize('test')` `cols = tbsize('test',2)`

`tbread`	Read from a table	`global name dob age`
		`vecs = {'name','dob','age'};`
		`cols = vecs;`
		`tbread('test',vecs,cols)`

`tbwrite`	Write to a table	`global name dob age`
		`name = {'John'}; dob = {'1-Jan-2000'}; age = NaN;`
		`vecs = {'name','dob','age'};`
		`tbwrite('test',vecs)`

`mym`	Submit an SQL command	`mym('create table test(name varchar(30),dob date,age double)')`
		`mym('insert into test values (''John'',''1-Jan-2000'', NULL)')`
		`[name,dob,age] = mym('select * from test')`

[8]Never delete system databases `mysql` and `information_schema`, or any of their tables.

3. Test drive.

Having completed the steps above, you can test the Matlab/MySQL connection by opening Matlab and entering

```
myopen('localhost','root','mypwd')
```

with `mypwd` replaced by your MySQL password. `mym.m` will try to connect to MySQL, and display the following message if it succeeds.

```
mYm v1.0.8, Copyright (C) 2006, Swiss Federal Institute of technology, Lausanne, CH
mYm comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
For details read the GPL license included with this distribution.
```

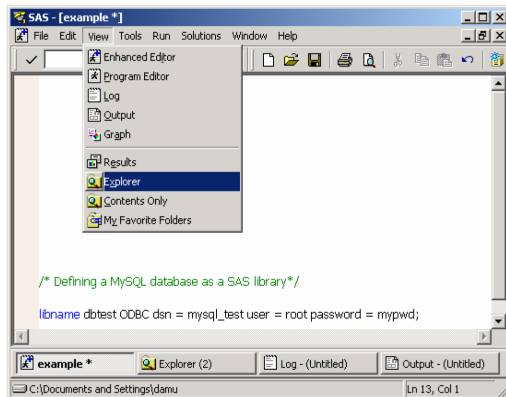
To check that MySQL is accessible to SAS - recall that by setting up a single MySQL data source, named `mysql_test`, we have granted SAS access only to database `test` - open SAS and submit

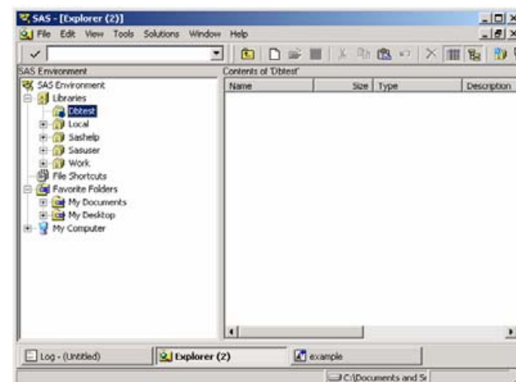
```
libname dbtest ODBC dsn = mysql_test user = root password = mypwd;
```

again replacing `mypwd` with the actual password. SAS will attempt an ODBC connection to `test`, and report the outcome in session log.

```
NOTE: Libref DBTEST was successfully assigned as follows:
      Engine:        ODBC
      Physical Name: mysql_test
```

At this point, you can switch to SAS Explorer window, and find `dbtest` in the list of the session's libraries.





The library is empty, as database `test` contains no tables. In the remainder of this section, we will fill the library with a dataset retrieved from WRDS and access it from Matlab, in the context of a simple exercise: counting how many firms from each SIC industry are found in the Compustat Industrial Annual file in each of the most recent five years[9].

We establish a remote connection to WRDS server

```
%let roland = wrds.wharton.upenn.edu 4016;
options pagesize = max comamid = TCP remote = wrds;
signon  username = _prompt_;
libname comp remote '/wrds/compustat/sasdata' server = wrds;
```

and select relevant data directly into a table in `test`.

[9] Compustat cognoscenti will take issue with variable `yeara`, fiscal year, being confused with the *calendar* year. We use it as a shortcut.

```
data mysql.example;
 set comp.compann (where = (yeara > 2000));
 if data6 > 0;   /* positive total assets required */
 keep yeara dnum gvkey;
 run;
```

As SAS log indicates, variable labels and formats - features that are specific to SAS - are lost in transition to MySQL[10].

```
NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
NOTE: There were 43404 observations read from the data set COMP.COMPANN.
NOTE: The data set DBTEST.EXAMPLE has 43404 observations and 3 variables.
```

(Another way in which ODBC libraries differ from 'native' SAS libraries is that datasets (i.e. tables) located in them have overwrite protection, and need to be deleted before a dataset's new version is created[11]. Third and most important difference is in the speed with which SAS reads from, and especially writes to, MySQL tables. As Table A1 of Appendix I demonstrates, ODBC-channeled read/writes are significantly slower than SAS's operations with 'native' datasets).

Switching to Matlab, we open database `test`

```
dbopen('test')
```

and verify that table `example` is visible to Matlab,

```
tblist

ans = 'example'
```

and has the expected structure and size.

```
[names, types] = tbattr('example')

names =  'DNUM'
         'GVKEY'
         'yeara'

types =  'double'
         'double'
         'double'

tbsize('example')

ans =   43404          3
```

We load contents of `example` into Matlab workspace, retaining variable name `gvkey`, but replacing `dnum` and `yeara` with the more intuitive `industry` and `year`.

```
[industry, gvkey, year] = mym('select * from example')
```

In this case, data fit into available memory, but had we worked with a (much) larger dataset and encountered an out-of-memory error, we might try to retrieve a subset of `example`, with a statement like

```
[industry, year] = mym('select industry, yeara from example')
```

or

```
[industry, gvkey, year] = mym('select * from example where yeara = 2005')
```

---

[10]See Appendix II, however.

[11]A database table accessible to SAS can be deleted with PROC DATASETS or via SAS Explorer's graphical interface. In Matlab, the task can be accomplished with `tbdrop`.

or

```
[industry, gvkey, year] = mym('select * from example limit 1000')
```

where the last variant would fetch `example`'s first thousand records.

(`limit` clause is not part of PROC SQL syntax, illustrating the point that SQL dialects of SAS and MySQL, albeit highly similar, are not identical. Should MySQL report a syntax error in a submitted query, your first trouble-shooting aid is the searchable online MySQL Reference Manual[12]. Watch out for functions that are SAS, not SQL, functions, such as `lag` or `intck`, and search for their MySQL counterparts. Although in some cases replacement may not be available - for instance, `intck` has a MySQL analog, `datediff`, but `lag` does not[13] - elsewhere MySQL may offer a function missing in SAS. Table 2 lists some of the functions available in MySQL).

Table 2. Selected MySQL functions.

Function	**Purpose**	**Example**
`year, month, day`	Extract date components	`year(date)`
`makedate`	Construct a date	`makedate(year,dayofyear)`
`date_add`	Increment/decrement a date	`date_add(date,interval 1 year)`
`datediff`	Count days between dates	`datediff(date1,date2)`
`date_format`	Format a date	`dateformat(date,'%W %M %Y')`
`date, time`	Extract datetime components	`date(datetime)`
`timestamp`	Construct a datetime	`timestamp(date,time)`
`addtime`	Increment/decrement a datetime	`addtime(date,time)`
`timestampdiff`	Measure interval btw datetimes	`timestampdiff('hour',dt1,dt2)`
`date_format`	Format a datetime	`dateformat(date,'%H:%i:%s')`
`char_length`	Measure length of a string	`char_length(name)`
`concat`	Concatenate strings	`concat(firstname,' ', lastname)`
`instr`	Find a substring	`instr(name,'John')`
`replace`	Replace a substring	`replace(name,'Bill', 'William')`
`substr, right, left`	Extract a substring	`substr(name,1,1)`

[12]You may find it useful to peruse the list of MySQL's reserved keywords, provided in Section 9.5 of MySQL 5.0 Reference Manual. Note that the list includes keyword `return`.
[13]To construct lagged values in MySQL, one uses a reflexive join. See the example in Appendix I.

Including variable gvkey in the working dataset, we had in mind the need to check for duplicate records, i.e. to make sure that one record corresponds to any given firm-year combination. The check can be coded like

```
I = unique(industry); ni = length(I);
Y = unique(year);     ny = length(Y);

for i = 1:ni
    for j = 1:ny
        x = gvkey(industry == I(i) & year == Y(j));
        y = unique(x);
        if length(x) ~= length(y)
           disp('Duplicate!')
        end
    end
end
```

but can be done in a simpler way with MySQL:

```
x = mym('select gvkey from example group by gvkey, yeara, dnum having count(*)> 1')

x =  Empty matrix: 0-by-1
```

(Table example is known not to contain any missing values of gvkey, yeara, or dnum, but if this were not the case - for example, if some values of yeara were missing - we would use is not null condition in where or having clause,

```
x = mym(['select gvkey from example where yeara is not null '...
         'group by gvkey, yeara, dnum having count(*)> 1')'])
```

to exclude unwanted cases).

Likewise, to produce a table of firm counts, with element (i,j) giving the number of firms of industry i on record in year j, we can use a 'pure Matlab' approach

```
N = zeros(ni,ny);
for i = 1:ni
    for j = 1:ny
        N(i,j) = sum(industry == I(i) & year == Y(i));
    end
end
```

or a 'mixed' one:

```
[yr,in,n] = mym('select yeara, dnum, count(*) from example group by yeara, dnum');
N = zeros(ni,ny);
for i = 1:ni
    for j = 1:ny
        x = n(in == I(i) & yr == Y(j));
        if ~isempty(x)
           N(i,j) = x;
        end
    end
end
```

On inspection, most of the code in the snippet above deals not with counting, but with reshaping the table of counts that was produced by the query in the first line. This suggests yet another (unorthodox) approach: have the query save its output to a table, and use SAS's PROC TRANSPOSE to reshape it[14].

Submitting the following line to Matlab,

```
mym('create table temp select yeara, dnum, count(*) from example group by yeara, dnum')
```

[14] N could be reshaped using long2wide.m, available from Matlab File Exchange.

we switch to SAS, verify that library dbtest now contains two datasets, example and temp - if Explorer window fails to refresh the library view, click on a different library, then again on dbtest - and run

```
proc transpose
 data = dbtest.temp
 out  = dbtest.counts_sas (drop = dnum _name_ _label_);
 by dnum;
 id yeara;
 run;
```

then return to Matlab and retrieve contents of counts_sas, this time using function tbread.

```
global N
N = zeros(ni,ny);
[vecs, cols] = deal(cell(5,1));
vecs = strcat('N(:,', int2str((1:5)'),')');
cols = cellstr(strcat('_', int2str(Y')));
tbread('counts_sas',vecs,cols)
```

Cell arrays vecs and cols contain the names of columns that are to be read from counts_sas - these are columns _2001, .. , _2005, created by PROC TRANSPOSE - and of the Matlab arrays that are to store incoming values. To populate a five-column matrix, we fill vecs with values 'N(:,1)', .. , 'N(:,5)'.

After the last code fragment is executed in Matlab, we need to replace NaN's in the counts matrix N with zeros,

```
N(isnan(N)) = 0;
```

which adds to the impression of the SAS-based approach as being more cumbersome than the rest. Our intention in presenting it was to demonstrate how data residing in a MySQL database can be nearly concurrently manipulated with Matlab and SAS, a capability that many WRDS users are likely to appreciate.

It remains to show how to transfer variables in Matlab workspace to a MySQL database. We conclude this exercise by saving N, the matrix of firm counts, as table counts_matlab of current database test. The operation requires two steps: creating an empty table of specified structure[15] with function tbadd

```
[cols, types] = deal(cell(5,1));
types(:) = {'double'};
cols = cellstr(strcat('N', int2str(Y'))); % Use column names N2001, N2002, etc.
tbadd('counts_matlab',cols,types)
```

and transferring N's contents into counts_matlab with tbwrite.

```
tbwrite('counts_matlab',vecs,cols)
```

We shut down Matlab's connection to MySQL with

```
myclose
```

[15]MySQL data types are discussed in Chapter 11 of MySQL 5.0 Reference Manual. In practice, one can limit attention to types double, date, and char(n) and varchar(n). (Argument n in the definition of character types char and varchar denotes the maximum allowed string length; to avoid having to 'resize' a character-type table column with , choose a value known to be sufficiently large). Interested readers may wish to explore the possibilities offered by MySQL's BLOB type (supported by mym.m) which allows saving a numeric array to a single *cell* of a MySQL table. (Consider saving data for various firms, or various sets of regression estimates, in distinct, indexed cells of a single MySQL table). We do not discuss BLOBs in this report, and refer to the helpful example in mym.m documentation.

4. Summary.

Matlab's inability to handle data volumes in excess of computer's memory resources, or access data stored in SAS's `sas7bdat` file format, such as those available from WRDS server, has severely limited the software's application by WRDS users, leading many to choose SAS as their primary programming tool. In this report, we suggest an approach that exploits SAS's edge at data retrieval, but breaks its 'hold' on high-volume data manipulation. Operations that to this point could only be done in SAS are now possible, and can be executed with reasonable efficiency, in Matlab. At the same time, the proposed approach offers a previously unavailable robust high-capacity facility for data transfer between Matlab and SAS, and thus serves a broader goal: enabling researchers familiar with both Matlab and SAS to use both packages in a single session, leveraging strengths of one with those of the other.

Appendix I. Evaluating MySQL's performance.

Advocating MySQL as a replacement for SAS, we have to disclose instances where its performance was found to be disappointing. Table A1 reviews a sequence of timing tests in which a group of data-manipulation tasks - retrieval, subsetting, sorting, merging, etc. - was performed in both MySQL and SAS. The tests involved a one-million-row subset of the CRSP monthly stock file, and employed a PC with a 1.8 GHz Pentium M processor and 512 MB of RAM, with a 'standard' configuration[16] of MySQL 5.0, SAS 9.1 and Matlab 7, running under Windows XP.

Table A1. Selected timing tests.

Task	Manipulating a SAS dataset with SAS	Manipulating a MySQL table with SAS	Manipulating a MySQL table with Matlab[17]
Retrieve data from WRDS	data sas.test; set crsp.msf (obs = 1000000); run; 3:30	data mysql.test; set crsp.msf (obs = 1000000); run; 20:26	(With mysql.test created) global cusip permno permco <..> vars = {'cusip','permno',<..>}; tbread('test',vars) 20:26 + 7:55
Describe data	proc contents data = sas.test; run; 0:00	proc contents data = mysql.test; run; 1:57	tbattr('test') tbsize('test') 0:02 + 1:10
Transfer data between MySQL and a SAS disk library	(SAS to MySQL) proc copy in = sas out = mysql; select test; run; 12:47	(MySQL to SAS) proc copy in = mysql out = sas; select test; run; 3:10	(Matlab workspace to MySQL) global cusip permno permco <..> vars = {'cusip','permno',<..>}; tbwrite('test',vars) 15:00
Transfer data to SAS WORK library	proc copy in = sas out = work; select test; run; 1:59	proc copy in = mysql out = work; select test; run; 3:14	n/a
Subset data: select rows	data sas.sub1; set sas.test (where = (vol = 0)); run; 1:03 data sas.sub2; set sas.test (where = (vol > 0)); run; 0:53	data mysql.sub1; set mysql.test (where = (vol = 0)); run; 3:10 data mysql.sub2; set mysql.test (where = (vol > 0)); run; 12:03	mym('create table sub1 select * from test where vol = 0') 1:18 mym('create table sub2 select * from test where vol > 0') 5:28
Concatenate subsets of data	data sas.test; set sas.sub1 sas.sub2; run; 0:45	data mysql.test; set mysql.sub1 mysql.sub2; run; 10:29	mym('insert into sub1 select * from sub2') tbrename('sub1','test') 5:21
Subset data: select columns	data sas.crop; set sas.test (keep = permno date); run; 1:25	data mysql.crop; set mysql.test (keep = permno date); run; 5:46	mym('create table crop select permno, date from test') 0:42

---

[16]Wishing to boost MySQL's speed, we briefly experimented with environment variables `key_buffer_size` and `table_cache`, increasing their values from 8,388,608 to 64,000,000, and from 256 to 512, respectively,

```
mym('set global key_buffer_size = 64000000')
mym('set global table_cache = 512')
```

but saw no improvement in the speed of the test join. Consult Section 7.5.2 of MySQL 5.0 Reference Manual for information on MySQL server parameters.

[17]We do not report results of exercises where SQL commands were submitted directly to MySQL, through MySQL Command Client window, as these were essentially identical to those obtained with Matlab and `mym.m`.

Sort data	proc sort data = sas.test out = sas.sort; by permno date; run; 3:19	proc sort data = mysql.test out = mysql.sort; by permno date; run; 12:51	mym('create table sort select * from test order by permno,date') 1:50
Compute summary statistics	proc sql; create table sas.stat as select date,mean(ret) from sas.test group by date; quit; 1:19	proc sql; create table mysql.stat as select date,mean(ret) from mysql.test group by date; quit; 3:24	mym('create table stat select date, avg(ret) from test group by permno, date') 0:15
Perform a join	proc sql; create table sas.join as select a.permno,a.date, a.prc,b.prc as lprc from sas.test a left join sas.test b on a.permno = b.permno and a.date > b.date and a.date < intnx('day',b.date,31); quit; 2:52	proc sql; create table mysql.join as select a.permno,a.date, a.prc,b.prc as lprc from mysql.test a left join mysql.test b on a.permno = b.permno and a.date > b.date and a.date < intnx('day',b.date,31); quit; 16:04	mym('create table join select a.permno,a.date,a.prc, b.prc as lprc from test a left join test b on a.permno = b.permno and b.date < a.date and a.date < date_add(b.date,interval 31 day)') 9:14:04
Perform a join (output to temp SAS dataset	proc sql; create table join <same as above> 3:17	proc sql; create table join <same as above> 10:04	n/a
Create an index	proc sql; create distinct index i on sas.test (permno,date); quit; 1:03	not possible	mym('create unique index i on test (permno,date)') 6:10
Perform a join (indexed)	<same as previous join> 3:44	not possible	<same as previous join> 40:06
Delete an index	proc sql; drop index i on sas.test; quit; 0:00	not possible	mym('drop index i on test') 6:40
Delete data	proc delete data = sas.test; run; 0:00	proc delete data = mysql.test; run; 0:03	tbdrop('test') 0:00

Generally, MySQL's performance is second to that of SAS, but the gap is tolerable, as MySQL's execution times are reasonably small. The crucial exception is the join exercise[18]: completed in just three minutes in SAS, it extended into nine hours in MySQL! Indexing the test dataset on join keys[19], variables `permno` and `date`, brought about a major improvement, but even so the join took more than ten times longer than if it were done in SAS. Based on this experience, we actually discourage use of MySQL for joins, and recommend that joins - and certainly large-scale joins - be done in SAS, with source data either remaining in MySQL, or transferred in SAS to a SAS library.

[18]Note that SAS offers a way of combining data from multiple tables that is complementary to the PROC SQL join: in DATA step, one can perform a 'simple' or a 'matched' merge, 'interleave' or 'concatenate' tables. (The difference between a DATA step match-merge and a PROC SQL inner join should be clearly understood). MySQL's repertoire is limited to joins and concatenation, with both options illustrated in Table A1.

[19]Defining an index takes time, and does not guarantee an improved speed - indeed, unless one uses `force index` option, one cannot be certain that an existing index will actually be used by a MySQL query. Refer to this SUGI white paper for an excellent (SAS-based) overview of indexing, and to Section 7.2.1 of MySQL 5.0 Reference Manual for a discussion of `explain select` statement. An `explain select` check proved to be instrumental in the reported exercise, leading us to discover that MySQL would not use index `i` if the constraint '`a.date > b.date and a.date < date_add(b.date, interval 31 day)`' were formulated as '`b.date < a.date < date_add(b.date, interval 31 day)`'.

Appendix II. Recovering SAS variable labels and formats.

As noted earlier, SAS variable labels and formats are lost when a SAS dataset is transferred through ODBC to a MySQL database. This is a nuisance, as variable labels contain useful information and are immensely helpful when variable names are uninformative or come in large numbers. Also, 'return trip' to SAS might at some point become necessary, and variable labels and formats needed - and have to be restored. Two SAS macros presented below offer help.

```
/* Save variable labels and formats in dataset DATA to dataset INFO */
%macro getLabelsAndFormats(data,info);
  %let p = %index(&data,.);
  %let n = %length(&data);
  %if &p = 0 %then %do;
      %let lib = work; %let dst = &data; %end;
  %if &p > 0 %then %do;
      %let lib = %substr(&data,1,&p-1);
      %let dst = %substr(&data,&p+1,&n-&p+1); %end;
  proc sql;
   create table &info
   as select name, label, format from dictionary.columns
   where lib  = upcase("&lib")
     and memname = upcase("&dst")
     and memtype = "DATA";
   quit;
%mend;

/* Apply variable labels and formats, saved by getLabelsAndFormats to
   dataset INFO, to dataset DATA */
%macro setLabelsAndFormats(data,info);
 proc sql noprint;
  select count(*) into :n from &info;
  select name   into :name1   - %sysfunc(compress(:name&n.))   from &info;
  select label  into :label1  - %sysfunc(compress(:label&n.))  from &info;
  select format into :format1 - %sysfunc(compress(:format&n.)) from &info;
  quit;
 data &data;
  set &data;
  %do i = 1 %to &n;
    label  &&name&i = "&&label&i";
    format &&name&i   &&format&i;
  %end;
  run;
%mend;
```

Macro `getLabelsAndFormats` extracts labels and formats from a SAS dataset. By directing the macro's output to a MySQL table, one makes labels immediately accessible to Matlab. Labels and formats can be stored in MySQL, and re-applied if the data are taken back to SAS, with macro `setLabelsAndFormats`. Consider the following example, where SAS dataset `test` is moved to a MySQL database, its labels saved in table `test_columns` and fetched to Matlab, and then taken back, with labels and formats restored.

```
proc copy
 in  = sas;
 out = mysql;
 select test;
 run;

%getLabelsAndFormats(sas.test,mysql.test_columns);

[name,label] = mym('select name, label from test_columns')     (in Matlab)

proc copy
 in  = sas;
 out = mysql;
 select test;
 run;

%setLabelsAndFormats(sas.test,mysql.test_columns);
```

Appendix III. Improving `tbwrite` speed.

Function `tbwrite` invokes SQL command `insert values` to add data to a MySQL table, with `buffer` rows of each `vecs` vector passed to the database in a single call. Choice of `buffer` (set to 1,000 by default) has a major impact on write speed, but the argument's optimal setting, which depends on the number and types of input vectors, is difficult to guess. In some cases, it may be worthwhile to try to identify the best choice of `buffer`, by selecting a subset of data, e.g. 1/100th or 1/20th of rows of each `vecs` vector, and repeatedly writing it to MySQL, varying the value of `buffer`. Recording the time of each run, one selects the best-performing `buffer` value and uses it for the 'full' write. Matlab code below illustrates the idea.

```
global x
x = rand(1,1e6);            % Original data
y = x(1:1e4);               % Write-test data
B = [10 100 1000 10000];    % BUFFER values to try
T = nan*B;                  % 'Stopwatch' times
for  i = 1:length(B)
     tbadd('write_test',{'y'},{'double'},'replace')
     tic
     tbwrite('write_test',{'y'},{},B(i))
     T(i) = toc;
end
bmax = B(T == min(T));      % Best BUFFER
tbadd('write_real',{'x'},{'double'},'replace')
tbwrite('write_real',{'x'},{},bmax)
```