Dynamic Query Tool Kit
For The Progress 4GL

Scott Auge

sauge@ amduus.com

License

The typical BSD license. In short, give credit in your docs and About menu. Don't be a troll and go suing me for freebie software.

```
/*
 * Removal of this header is illegal.
 * Written by Scott Auge scott_auge@yahoo.com sauge@amduus.com
 * Copyright (c) 2006 Amduus Information Works, Inc.  www.amduus.com
 * Copyright (c) 2006 Scott Auge
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by Amduus Information Works
 *      Inc. and its contributors.
 * 4. Neither the name of Amduus Information Works, Inc. nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY AMDUUS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AMDUUS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 */
```

Contributors

Dayne May daynem @ linx.com.au
Coding for handling multi-DB applications.

Tony Suslovich tony.suslovich @ gmail.com
Coding for providing RECID and ROWID values for buffers in query

Introduction

With Progress Version 9.1, a very powerful feature was added to the language – dynamic queries.

Great as they are – it takes a lot of work to make use of this new feature. It is the purpose of this tool kit to aid with simplifying dynamic queries for the developer.

The tool kit provides a set of functions that can be used to open, navigate, extract data from, and close dynamic queries in a straight forward and easy to use manner. In my opinion it is the way it should have been done in the first place – but hey – gives me a chance to shine with my programming.

This tool kit should allow you to work in GUI, CHUI, WWW, and CLI based environments effortlessly.

About Scott Auge

Scott Auge has been working with Progress Technologies for over ten years (since version 6.) He focuses on UNIX and Web based software development. He is the founder and president of Amduus Information Works, Inc. He can be reached at sauge@ amduus.com (without the space.)

About Amduus Information Works, Inc.

Amduus Information Works, Inc. is a software development and placement company. We work with Progress technologies as well as PHP, PostgreSQL, and MySQL technologies. We look forward to hearing from you! Find out more about Amduus at http://www.amduus.com.

Example Code For Reading Dynamic Queries

The following code is a simple example of setting up the query and using the various routines available in the tool kit to work the query[1].

I think you will find it far easier to read and work than the more raw level 4GL coding required for something like this.

```
/*************************************************************/
/* Test One : A good query                                   */
/*************************************************************/

{dyntoolkit.i}

DEFINE VARIABLE h AS HANDLE NO-UNDO.

ASSIGN h = dyn_open("FOR EACH Job NO-LOCK").

DISPLAY cDyn_ErrCode cDyn_ErrMsg FORMAT "x(30)".

REPEAT:

  dyn_next(h).
  IF dyn_qoe(h) THEN LEAVE.

  DISPLAY dyn_getvalue(h, "Job.JobID")
          dyn_getvalue(h, "Job.Name")
          dyn_getvalue(h, "Job.Priority")
          dyn_getvalue(h, "Job.ErrCode").

END.

dyn_close(h).
```

Here is another sample of code. It helps illustrate how to handle errors in the

1 The dyn_next() call makes it appear one is skipping the first record, but it does not. This is on par to working with dynamic queries outside of the tool kit also.

query.

Notice one can check the value of the handle returned – if it is the unknown value, then an error occurred in processing of the query.

Additional information can be found in the `cDyn_ErrCode` and `cDyn_ErrMsg` varables. The code is a three digit number. The message is more human friendly and includes the query that was attempted.

```
/*************************************************************/
/* Test One : A bad query                                    */
/*************************************************************/

{dyntoolkit.i}

DEFINE VARIABLE h AS HANDLE NO-UNDO.

ASSIGN h = dyn_open("FOR EACH NoTable NO-LOCK").

DISPLAY cDyn_ErrCode cDyn_ErrMsg  FORMAT "x(30)".

IF h = ? THEN STOP.

REPEAT:

  dyn_next(h).
  IF dyn_qoe(h) THEN LEAVE.

  DISPLAY dyn_getvalue(h, "Job.JobID")
          dyn_getvalue(h, "Job.Name")
          dyn_getvalue(h, "Job.Priority")
          dyn_getvalue(h, "Job.ErrCode").

END.

dyn_close(h).
```

See the API list for more information about error handling.

Using the Dynamic Toolkit On An Existing Query

Here is another example. We define and open the query in one program, but we actually want to use it in another. So we pass the handle into the secondary external program and then apply it to `dyn_qryinfo()` to rebuild our toolkit's data structures.

```
{dyntoolkit.i}


DEFINE VARIABLE h AS HANDLE NO-UNDO.

ASSIGN h = dyn_open ("FOR EACH JOB NO-LOCK").

RUN test2.p (INPUT h).

dyn_close(h).
```

Note in the above program we simply create the query and then pass it's handle into an external program (or persistent procedure.)

Inside that external program, before we can use the tool kit's APIs, we will need to call dyn_qryinfo() first. After that is called, you can use the tool kit's APIs as if the query was made in the program.

```
{dyntoolkit.i}


DEFINE INPUT PARAMETER h AS HANDLE NO-UNDO.

/*************************************************************************/
/* Prep data structures from passed in handle for use with dyntoolkit APIs  */
/*************************************************************************/
```

```
dyn_qryinfo(h).

/*************************************************************************/
/* Do a nice little loop to show the idea works and dyn_qryinfo() works.    */
/*************************************************************************/

REPEAT:

  dyn_next(h).
  IF dyn_qoe(h) THEN LEAVE.

  DISPLAY dyn_getvalue(h, "Job.JobID")
          dyn_getvalue(h, "Job.Name")
          dyn_getvalue(h, "Job.Priority")
          dyn_getvalue(h, "Job.ErrCode").

END.
```

Updating The Results Of A Dynamic Query

There are a set of routines to help you update the results of a dynamic query. One set assumes you, the programmer KNOW the data type of the field to be assigned and expects you to send in those types. The other set will accept a `CHARACTER` version of the data, and upon using the fields `DATA-TYPE`, cast the value into the right data type and assign the field.

WARNING: Updating the field(s) that determine the order of the dynamic querywill send the query into la-la land. See example program commenting for more information.

Here is an example of a program that will accept values as a type `CHARACTER` and figure out based on the field's type definition how to cast the value:

```
{dyntoolkit.i}
```

```
DEFINE VARIABLE cRCS AS CHARACTER INIT "$Id: testwrt1.p,v 1.1
2006/06/30 05:13:54 sauge Exp sauge $" NO-UNDO.

DEFINE VARIABLE h AS HANDLE NO-UNDO.

DEFINE VARIABLE c AS CHARACTER NO-UNDO.


/*************************************************************/
/* Test One : A good query in DB 1 with a write to the buffers  */
/*************************************************************/

HIDE ALL.

ASSIGN h = dyn_open("FOR EACH Job EXCLUSIVE-LOCK").

DISPLAY cDyn_ErrCode cDyn_ErrMsg FORMAT "x(30)".

DISPLAY h:NUM-RESULTS COLUMN-LABEL "NumResults".

/*************************************************************/
/* Note with an exclusive lock, we need TRANSACTION for our   */
/* looping.                                                   */
/*************************************************************/

REPEAT TRANSACTION:

  dyn_next(h).
  IF dyn_qoe(h) THEN LEAVE.

  DISPLAY dyn_getvalue(h, "Job.JobID")
          dyn_getvalue(h, "Job.Name")
          dyn_getvalue(h, "Job.Priority")
          dyn_getvalue(h, "Job.ErrCode").

  /*********************************************************/
  /* Check our write ability                                */
  /*********************************************************/

  /*********************************************************/
  /* In this table, the JobID is used to order the query.  When */
  /* updating this value, the query tends to spin off to la-la  */
  /* land. Reading it works just fine so no worries there.      */
  /* This is an example of code that will NOT work.             */
  /* You will need to deal with this problem programmatically.  */
  /* JobID is a primary unique index in the Job table.          */
  /*********************************************************/
```

```
  /* dyn_setc(h, "Job.JobID", STRING(TIME)).*/



  /************************************************************/
  /* These are indexed, but not involved in the ordering of   */
  /* the query - so there is no problem.                      */
  /************************************************************/

  /************************************************************/
  /* Note unlike testwrt.p, we are using dyn_set which will   */
  /* figure out the data type according to the field and at-   */
  /* tempt to type cast according to that for setting the value.*/
  /************************************************************/

  dyn_set(h, "Job.STATE", "CRAIG").
  dyn_set(h, "Job.Priority", STRING(TIME)).
  dyn_set(h, "Job.ALogical", "YES").
  dyn_set(h, "Job.PrcStartDate", STRING(TODAY)).
  dyn_set(h, "Job.ADecimal", STRING(4.22)).

  PAUSE 1.

END.

dyn_close(h).
```

One of the first things to notice, is that when updating the values in a dynamic query, you need to query with an `EXCLUSIVE-LOCK`.

In addition, you need to wrap the transition through the records within a `TRANSACTION` as done in the above `REPEAT` loop.

The actual updating is done by the `dyn_set()` function. This function will accept the value to update the field with as a character. Then depending on the data type of the field according to it's schema definition, cast that value into the actual value. This function is useful when you don't know what the data type of

an unknown field is at run-time.

Here is another example program should you already know the types that fields are:

```
{dyntoolkit.i}

DEFINE VARIABLE cRCS AS CHARACTER INIT "$Id: testwrt.p,v 1.1 2006/06/30
04:41:25 sauge Exp $" NO-UNDO.

DEFINE VARIABLE h AS HANDLE NO-UNDO.

DEFINE VARIABLE c AS CHARACTER NO-UNDO.


/*************************************************************/
/* Test One : A good query in DB 1 with a write to the buffers  */
/*************************************************************/

HIDE ALL.

ASSIGN h = dyn_open("FOR EACH Job EXCLUSIVE-LOCK").

DISPLAY cDyn_ErrCode cDyn_ErrMsg FORMAT "x(30)".

DISPLAY h:NUM-RESULTS COLUMN-LABEL "NumResults".

/*************************************************************/
/* Note with an exclusive lock, we need TRANSACTION for our   */
/* looping.                                                  */
/*************************************************************/

REPEAT TRANSACTION:

  dyn_next(h).
  IF dyn_qoe(h) THEN LEAVE.

  DISPLAY dyn_getvalue(h, "Job.JobID")
          dyn_getvalue(h, "Job.Name")
          dyn_getvalue(h, "Job.Priority")
          dyn_getvalue(h, "Job.ErrCode").

  /*************************************************************/
```

```
  /* Check our write ability                                    */
  /*************************************************************/

  /*************************************************************/
  /* In this table, the JobID is used to order the query.  When */
  /* updating this value, the query tends to spin off to la-la  */
  /* land. Reading it works just fine so no worries there.      */
  /* This is an example of code that will NOT work.             */
  /* You will need to deal with this problem programmatically.  */
  /* JobID is a primary unique index in the Job table.          */
  /*************************************************************/

  /* dyn_setc(h, "Job.JobID", STRING(TIME)).*/



  /*************************************************************/
  /* These are indexed, but not involved in the ordering of     */
  /* the query - so there is no problem.                        */
  /*************************************************************/

  dyn_setc(h, "Job.STATE", "SCOTT").
  dyn_seti(h, "Job.Priority", TIME).
  dyn_setl(h, "Job.ALogical", YES).
  dyn_setd(h, "Job.PrcStartDate", TODAY).
  dyn_setf(h, "Job.ADecimal", 3.22).

  PAUSE 1.

END.

dyn_close(h).
```

You can see the program is very simular to the first one – only this time we are using `dyn_set*()` functions appropriate to the data type of the value to set the field to.

API List

The APIs are listed more in an order of use than in alphabetical order.

```
FUNCTION dyn_open RETURNS HANDLE (INPUT cQry AS CHARACTER)
```

Use the dyn_open () api to open up a query. Simply give the FOR EACH or FIND styles available in dynamic queries (see Progress Documentation) as an argument. The function will return a value you need to place into a handle variable.

```
FUNCTION dyn_next RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_next() api to move to the next row available in the query. It requires the handle sent from the dyn_open() api to identify the query you wish to move around in.

Be sure to check the dyn_qoe() api to determine if you have moved past the beginning or the end of the query record set.

```
FUNCTION dyn_prev RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_prev() api to move to the prev row available in the query. It requires the handle sent from the dyn_open() api to identify the query you wish to move around in.

Be sure to check the dyn_qoe() api to determine if you have moved past the beginning or the end of the query record set.

```
FUNCTION dyn_last RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_last() api to jump to the last row available in the query. It requires the handle sent from the dyn_open() api to identify the query you wish to move around in.

Be sure to check the dyn_qoe() api to determine if you have moved past the beginning or the end of the query record set.

```
FUNCTION dyn_first RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_first() api to jump to the first row available in the query. It requires the handle sent from the dyn_open() api to identify the query you wish to move around in.

Be sure to check the dyn_qoe() api to determine if you have moved past the beginning or the end of the query record set.

```
FUNCTION dyn_qoe RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_qoe() api to determine if you have run out of records in the query result at the end, or have tried to move past the beginning in a dyn_prev() call. It requires the handle sent from the dyn_open() api to identify the query you wish to move around in.

```
FUNCTION dyn_getvalue RETURNS CHARACTER (INPUT hQryHndl AS HANDLE,
                                         INPUT cTblFld AS CHARACTER)
```

Use the dyn_getvalue() api to pull out the value from a table.field combination found in the query. You will need to send in an argument of the table name and field in table.field notaion; as well you will need to send in the handle to the query you wish to pull the data from.

```
FUNCTION dyn_close RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

Use the dyn_close() api to close the query and to deallocate all the dynamicly created objects from the dyn_open() api call. **If you do not do this, YOU WILL GET MEMORY LEAKS.**

```
FUNCTION dyn_gettables RETURNS CHARACTER (INPUT cQry AS CHARACTER)
```

Given the text of a query, return the tables that are in that query in a comma delimited list.

```
FUNCTION dyn_numresults RETURNS INTEGER (INPUT hQryHndl AS HANDLE)
```

Given a query handle, return the number of items in the query's result set.

```
FUNCTION dyn_dump RETURNS LOGICAL (INPUT cFileName AS CHARACTER)
```

Internal use mostly – dump out the internal temp-table of values to cFileName. Documented for completeness.

```
FUNCTION dyn_qryinfo RETURNS LOGICAL (INPUT hQryHndl AS HANDLE)
```

The tool kit uses a temp-table to match names to allocated objects. When you call out to an external procedure that is unaware of this temp-table – the tool kit APIs will not work. The dyn_qryinfo() routine will rebuild this temp table in the external procedure called from the given query handle. This way, you can pass the query handle as an argument, rebuild the tool kit's internals with this API and then use the other APIs as if the query were born in the given file.

```
FUNCTION dyn_set RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
```

```
                                    INPUT cTblFld AS CHARACTER,
                                    INPUT cText AS CHARACTER):
```

Sets the given field to the given `cText` value. This routine will figure out the type to cast the character `cText` into before assigning it to the field value.

LOGICAL types should be an element of {Y,N,YES,NO,TRUE,FALSE}.

```
FUNCTION dyn_setc RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
                                    INPUT cTblFld AS CHARACTER,
                                    INPUT cText AS CHARACTER):
```

Assigns a CHARACTER type field to the cText value.

```
FUNCTION dyn_seti RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
                                    INPUT cTblFld AS CHARACTER,
                                    INPUT iVal AS INTEGER):
```

Assigns the given INTEGER type field to the INTEGER iVal.

```
FUNCTION dyn_setf RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
                                    INPUT cTblFld AS CHARACTER,
                                    INPUT fVal AS DECIMAL):
```

Assigns the given DECIMAL type field to the DECIMAL fVal.

```
FUNCTION dyn_setl RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
                                    INPUT cTblFld AS CHARACTER,
                                    INPUT lVal AS LOGICAL):
```

Assigns the given LOGICAL type field to the LOGICAL lVal.

```
FUNCTION dyn_setd RETURNS LOGICAL (INPUT hQryHndl AS HANDLE,
                                   INPUT cTblFld AS CHARACTER,
                                   INPUT dVal AS DATE):
```

Assigns the given DATE type field to the DATE dVal.

```
FUNCTION SET_LOGICAL RETURNS LOGICAL (INPUT cText AS CHARACTER):
```

When cText is ?, it will return a ?. When cText is an element of {Y,YES,TRUE} it will return TRUE, else FALSE.

```
FUNCTION dyn_listfields RETURNS CHAR (INPUT hQryHndl AS HANDLE,
                                      INPUT cTblName AS CHAR):
```

Provide a comma seperatated list of field names for the given table in the given query.

```
FUNCTION dyn_listtables RETURNS CHAR (INPUT hQryHndl AS HANDLE):
```

For the given open query, provide a comma seperated list of tables in the query.

```
FUNCTION dyn_numfields RETURNS INTEGER (INPUT hQryHndle AS HANDLE,
INPUT cTbleName AS CHARACTER):
```

List the number of fields in the given table in the given open query.

```
FUNCTION dyn_numtables RETURNS INTEGER (INPUT hQryHndle AS HANDLE):
```

List the number of tables in the given open query.

```
FUNCTION dyn_fieldtype RETURNS CHARACTER
(INPUT hQryHndl AS HANDLE,
INPUT cTblFld AS CHARACTER):
```

Given the query and the table.field in the query, return the PROGRESS DATA-TYPE for the field.

```
FUNCTION dyn_fieldhdl RETURNS HANDLE (INPUT hQryHndl AS HANDLE,
                                      INPUT cTblFld AS CHARACTER):
```

Given the query and the table.field in the query, return the PROGRESS field handle.

```
FUNCTION dyn_tablehdl RETURNS HANDLE (INPUT hQryHndl AS HANDLE,
                                      INPUT cTbl AS CHARACTER):
```

Given the query and the table in the query, return the PROGRESS table handle.

```
FUNCTION dyn_getvalue_raw RETURNS RAW (INPUT hQryHndl AS HANDLE,
                                       INPUT cTblFld AS CHARACTER):
```

Given the query and the table.field in the query, return the PROGRESS field as a raw (in case you are working with BLOBs or other RAW types.)

```
FUNCTION dyn_getvalue_rowid RETURNS ROWID (INPUT hQryHndl AS
HANDLE, INPUT cTblName AS CHARACTER):
```

Given the query and the table name, return the ROWID for that record.

```
FUNCTION dyn_getvalue_recid RETURNS RECID (INPUT hQryHndl AS
HANDLE, INPUT cTblName AS CHARACTER):
```

Given the query and the table name, return the RECID for that record.

Additional Programs

```
dyn_findinschema.p (INPUT cToken, INPUT lIsTable).
```

This program is called by the dyn_gettables() function in dyntookit.i WHEN the &USEMULTI preprocessor is set to YES. It is used to seach across multiple connected Dbs to determine if the token is a table or not. This requires a TEMPDB alias to be created before use.

Error Handling

If the tool kit encounters errors, it will populate two special variables named cDyn_ErrCode and cDyn_ErrMsg. The variable cDyn_ErrCode is a character of three digits containing the code number. The variable cDyn_ErrMsg will contain a more human friendly version of the message.

Values and meanings found in cDyn_ErrCode:

Code	***Purpose***
"000"	No Errors
"100"	Could not prepare query
"101"	Could not open query
"102"	Could not determine tables

Performance Considerations (Single and Multi-DB applications)

The code is written by default with the idea of being connected to multiple Dbs at a time.

For the performance addicts out there, it is possible to skip a loop and a `RUN` by setting the `USEMULTI` preprocessor to `NO` *only* when you are connected to a single DB. By default, this is set to `YES`. In my opinion the performance is no different between the modes.

When switching values on this preprocessor – you will be required to re-compile your code. The preprocessor instructions are designed to compile one set of code or the other – this is NOT a run-time option.

See `dyntoolkit.i` for more information.

Errata

Can I have more than one query working in the code?

Yes, but each query will need it's own handle.

Can I send the query handle to a external procedure and use the toolkit there?

Yes, but you will need to use dyn_qryinfo() before trying to use the toolkit's APIs in the external procedure.. If you look at the code in `dyntoolkit.i`, you will see that it is dependent upon a temp-table to help it out with tracking allocated objects and their names.

See the coding example for more information.

Can I send the query handle into an internal procedure and use the toolkit there?

Yes. The above mentioned temp-table is scoped globally to the procedure file.

Can I use the tool kit against a query that is made outside of the tool kit? Like on existing code?

Yes. You will need to use the dyn_qryinfo() function first so the toolkit's APIs are prepped. There is nothing special about the queries the toolkit uses – they are exactly the same as your own Progress 4GL coded dynamic queries.