GT.M for the Python Programmer

Chapter 1: Introduction

About GT.M

GT.M is a very high-performance, highly scalable schema-less NoSQL database.

It is based on a database technology that includes two components:

- a built-in native language (known as the Mumps or M language)
- a hierarchical, schema-less database management system

This paper will only focus on the latter component, the database within GT.M, and will summarise how it can be used by a Python developer. If you're interested in the native language, other resources such as http://gradvs1.mgateway.com/download/extreme1.pdf can be consulted.

GT.M is available as a Free Open Source version which, of course, makes it a very attractive proposition for the Python developer, particularly since the Open Source version contains the full range of GT.M features that allows it to scale up from a Netbook to a multi-site, multi-processor enterprise-scale system.

This paper will assume that you've made use of the M/DB installer (http://gradvs1.mgateway.com/main/index.html?path=mdb/mdbDownload) which is probably the simplest, easiest and most pain-free way of getting up and running with GT.M. You'll also find that the M/DB installer includes our *m_python* binding library that allows you to access the GT.M database from Python.

You can use the M/DB installer with any Ubuntu (or other Debian) 32-bit Linux system, ideally a "virgin" Ubuntu 9.10 new install on either a dedicated server or, more likely, a virtual machine or Amazon EC2 instance. The link above for the M/DB installer includes full instructions for getting an M/DB Appliance up and running, a process that should take only a matter of a few minutes.

The GT.M Database

The really important heart of GT.M is its data storage mechanism. This is based on what are known as Persistent Global Variables, more commonly known simply as Globals. Globals are an incredibly simple concept, and yet incredibly powerful.

Globals are a primitive structure but they provide a data storage mechanism that is fast to use, lean, mean and totally malleable. Essentially globals are persistent, multi-dimensional sparse arrays, where the array subscripts can be either numeric or strings. In fact they are similar in concept to the Python Dictionary, except that they persist on disk rather than in memory.

The next two chapters of this paper examine Globals in more detail.

Chapter 2 will summarise the basics of Globals. Chapter 3 focuses on their use in terms with which a RDBMS/SQL programmer will be familiar. If you wish, skip to Chapter 3 now and return to Chapter 2 to familiarise yourself with the basics later.

First, a quick primer on how to run GT.M from your newly installed M/DB Appliance.

In the M/DB Appliance, a complete GT.M instance has been installed and configured in the directory path ***/usr/local/gtm/ewd*** [Note: if you are running the M/DB Appliance in an EC2 instance with an EBS volume, the directory path to use will be ***/mdb/ewd***]

You can access GT.M via its native language by doing the following:

```
cd /usr/local/gtm/ewd
$gtm
```

You'll now see the GT.M shell prompt:

```
GTM>
```

This is an interactive shell, much like the Python shell. Although this paper will focus on the use of Python with GT.M, you'll find that certain things are most easily done using the GT.M shell, for example examining the raw structure of the Globals that you'll be creating,

Ideally, then, have two Linux processes running simultaneously, one with the GT.M shell open, the other with the Python shell running.

To exit the GT.M shell, simply type ***H*** followed by the Enter key (all GT.M's native language commands can be reduced to a single letter, and H is the abbreviation for HALT. All commands are case insensitive, so ***h*** will do).

To allow easy access to GT.M's globals from Python, the M/DB Appliance includes a pre-built module in */usr/local/gtm/ewd* named ***gtm.py***. If you import this, it will load the *m_python* binding library and link it to the GT.M instance for you. For example:

```
ewd@mdb:~$ cd /usr/local/gtm/ewd
ewd@mdb:/usr/local/gtm/ewd$ python
Python 2.5.1 (r251:54863, Mar  7 2008, 04:10:12)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import gtm
>>>
```

To run the examples described in Chapters 2 and 3, it will be assumed that you're running in the Python shell with the *gtm.py* module loaded as shown above or using your own module (see below).

To check everything in your M/DB Appliance is correctly running and configured for Python, try running the test function in *gtm.py* (having first run *import gtm*):

```
>>> gtm.test()
M/Gateway Developments Ltd. - m_python: Python Gateway to M - Version 2.0.41
^robtest2(1,"a","b") saved
Reading back value:
Rob Tweed
^robtest2(1,"a","b") killed
If you see this, the test was successful and m_python is working
>>>
```

Once you're familiar with using GT.M and the *m_python* syntax to access GT.M's globals, you can build your own Python modules. You'll find a number of pre-built examples that you can use as a template for your own, eg *mdb.py* (which allows the M/DB database to be accessed from Python) and *ewd.py* (which allows the EWD Ajax framework to be used with Python). Essentially you'll need the following at the start of your module to create the correct working environment for the *m_python* functions:

```
import m_python

gtm_dist    = "/usr/local/gtm"
gtmroutines = "/usr/local/gtm/ewd"
gtmgbldir   = "/usr/local/gtm/ewd/mumps.gld"

gtmci       = "/usr/mgwsi/m/gtm/zmgwsi.ci"
use_gtm_api = m_python.m_bind_gtm_server(0, gtm_dist, gtmci,
gtmroutines, gtmgbldir, "", "", "")
```

OK, we're ready to delve deeper!

Chapter 2: Globals: the Basics

GT.M uses Globals as the basis of its storage mechanism. Applications such as M/DB and M/DBX layer more “conventional” views onto the core Global construct, eg M/DB layers the Amazon SimpleDB database structure on top of GT.M's globals, whilst M/DBX layers a persistent XML DOM database structure on top of GT.M's globals.

M/DB and M/DBX are therefore good examples of how the basic Global storage in GT.M can be used to model many different logical database structures. Furthermore, different applications can be run at the same time in a GT.M database, each one simultaneously giving a different logical view of the underlying simple Global-based database. GT.M has been described as the “Swiss Army Knife of Databases” for this reason.

So what are Globals?

Put simply, a Global is a persistent, sparse, dynamic, multi-dimensional array, containing a text value. Actually GT.M allows the use of both persistent and non-persistent multi-dimensional arrays, the latter known as “local arrays”.

Somewhat unusually, Globals allows the use of both numeric and textual subscripting. So in GT.M, you can have an array such as:

Employee(company,country,office,employeeNumber) = employeeDetails

eg

Employee(“MGW”,”UK”,”London”,1) = “Rob Tweed`Director`020 8404 1234”

In this example, the data items that make up the employee details (name, position, telephone number) have been appended together with the back-apostrophe character as a delimiter. GT.M does not impose any controls or rules over how you construct your data structures : there’s no schema or data dictionary for describing your database records. This gives you incredible flexibility and speed of development. You can arbitrarily assign one or more data delimiters to break the text value of an array record into a number of “fields”. The total string length that can be assigned to a single array record is configurable, but in the default GT.M configuration (as used by the M/DB installer) it is up to 4k. The string length that is actually stored is variable, and you can see that by using a delimiter character, individual fields are variable length also. This makes GT.M's Globals a very efficient data storage mechanism : there’s almost no disc space wasted holding empty, unused space characters.

Now in the example above, the employee record will be held in what is known as a "Local Array". If you were to exit from your GT.M session, the array would disappear, just like a PHP array once the page or session has gone.

Now here's the fun bit. To store this employee record permanently to disc, ie as a Global, just add a "^" in front of the array name:

^Employee(company,country,office,employeeNumber) = employeeDetails

eg

^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 1234"

That's all there is to it!

Setting a Global Value

So, to create such a global node using the GT.M's native language, you would use the command "set", ie:

set ^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 1234"

This command can be abbreviated down to a single letter (in upper or lower case:

s ^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 1234"

Now when you exit from the GT.M environment, the record will persist on disc permanently. When you come back at any time in the future, you can retrieve the record straight off the disc by using the global reference:

^Employee("MGW","UK","London",1)

Now let's look at how you can do that in Python. In your Python shell, type the following and press Enter::

```
gtm.m_python.m_set(0, "^Employee", "MGW", "UK", "London", 1, "Rob
Tweed`Director`020 8404 1234")
```

You should get a response of ``. The syntax of the m_python *m_set()* function is:

- parameter 1: use 0 to use the current GT.M instance
- parameter 2: the Global name. Make sure it starts with a ^
- parameters 3...n-1: the Global subscripts, as many as required
- last parameter: the data value for the global

Let's check that it has really stored your Global. Go to your GT.M shell and type:

```
GTM> zwr ^Employee
```

This will write out the entire contents of the ^Employee global, so you should see:

```
^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 1234"
```

Getting a Global Value

To get a value back from a Global, use the m_python m_get() function, eg:

```
record = gtm.m_python.m_get(0, "^Employee", "MGW", "UK",
"London", 1)
```

The syntax of the m_python *m_get()* function is:

- parameter 1: use 0 to use the current GT.M instance
- parameter 2: the Global name. Make sure it starts with a ^
- parameters 3...n-1: the Global subscripts, as many as required

For example:

```
>>> record = gtm.m_python.m_get(0, "^Employee", "MGW", "UK", "London", 1)
>>> record
'Rob Tweed`Director`020 8404 1234'
>>>
```

Globals can have an arbitrary number of subscripts, and the subscripts can be any mixture of text and numbers (real or integers). String subscripts must be surrounded by double quotes, while for numeric subscripts, quoting is optional.

Note that GT.M has a maximum subscript length of 247 characters. This limits the number of subscripts you can practically have in a Global reference. The longer the subscript values you use, the fewer the total number of subscripts you'll be able to specify. Whilst this can sound somewhat limiting, in practice it turns out not to be a practical issue for reasons that will hopefully become clear.

Deleting a Global Value

To delete a Global node, use the m_python m_kill() function, eg:

```
record = gtm.m_python.m_kill(0, "^Employee", "MGW", "UK",
"London", 1)
```

The syntax of the m_python *m_kill()* function is identical to the *m_get()* function:

- parameter 1: use 0 to use the current GT.M instance
- parameter 2: the Global name. Make sure it starts with a ^
- parameters 3...n-1: the Global subscripts, as many as required

For example:

```
>>> gtm.m_python.m_kill(0, "^Employee", "MGW", "UK", "London", 1)
''
>>>
```

If you now check the global from the GT.M shell, you'll find that it no longer exists:

```
GTM>zwr ^Employee
%GTM-E-GVUNDEF, Global variable undefined: ^Employee

GTM>
```

That's Really It!

That's fundamentally all there is to Globals. The real trick is how to make such a primitive data structure work for you. That's what the rest of this paper will do. We'll attempt to do this in such a way that the relational database programmer can understand the equivalent techniques and representations that a GT.M database practitioner would use.

Remember that there's nothing in GT.M's globals that will enforce a particular database design methodology, and its up to you to add the controls and checks that will ensure your database is logically consistent and error-free. That means you'll be doing a lot of work that a more conventional DBMS would otherwise do for you, but you'll soon find that you can automate the most repetitive tasks, and make light work of the management of a Global-based database.

Creating a simple multi-level hierarchy

You may have multiple levels of subscripting simultaneously in your globals, eg:

```
^Employee("MGW","UK","London") = 2
^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 1234"
^Employee("MGW","UK","London",2) = "Chris Munt`Director`01737 371999"
```

Here we're specifying the number of employees at a given office at the third level of subscripting, and the actual employee record at the fourth level.

Indexes and/or linkages between different globals are for the programmer to define explicitly. ***GT.M provides no automatic indexing or cross-linking of globals for you.***

Hence, we could have a telephone number global that acts as an index to the employee global, eg:

^EmployeeTelephone("020 8404 1234") = "MGW`UK`London`1"
^EmployeeTelephone("01737 371999") = "MGW`UK`London`2"

In this example, the data value stored against each telephone number holds the subscripts for the associated employee record, concatenated together. By knowing the telephone number, all we'd have to do is break the data value apart using the back apostrophe as a delimiter, and we could retrieve the main employee record.

For example:

```
telNo="020 8404 1234"
indexData = m_python.m_get(0,"^EmployeeTelephone",telNo)
fs = indexData.split("`")
rec = m_python.m_get(0,"^Employee",fs[0],fs[1],fs[2],fs[3])
employeeName = rec.split("`")[0]
```

One of the great things is that nothing in GT.M has to be pre-declared. You decide when and how to create, modify or delete global records – it's all automatically and dynamically handled for you. You can add further pieces to your global at any time without any need for declarations whatsoever. If you want to start using another global, just start using it and it will be created dynamically for you.

Setting, Getting and Deleting Globals

To summarise, the *m_python* gateway allows you to manipulate GT.M Globals from within Python. Globals are created using the *m_set()*, retrieved using the *m_get()* function and deleted using the *m_kill()* function.

1) Creating a global record:

```
m_python.m_set(0, "^Employee", "MGW", "UK", "London", 1, "Rob
Tweed`Director`020 8404 1234")
```

This creates the global reference, saving the record to disc.

2) Retrieving a global record

```
rec = m_python.m_get(0, "^Employee", "MGW", "UK", "London", 1)
```

This retrieves the specified global and places the data value into the Python variable *rec*

3) Deleting a global record:

```
m_python.m_kill(0, "^Employee", "MGW", "UK", "London", 1)
```

This permanently and immediately deletes the specified global record from disc. *Be very careful with the Kill command* – it's both extremely simple to use and incredibly dangerous. If you specify fewer subscripts, all lower-level subscripts will be deleted. If you specify no subscripts at all, the entire global will be deleted, eg:

```
m_python.m_kill(0, "^Employee", "MGW", "UK")
```

This will delete all records for all UK offices

```
m_python.m_kill(0, "^Employee")
```

This will delete the <u>*entire*</u> ^Employee global, ***immediately, permanently and irretrievably***, unless you'd backed it up.

Traversing a Global

One of the most frequent things you need to do is traverse some or all of a global. For example, let's say you want to manipulate all the employee records, perhaps to display a list so that the user can select one of them, or to count them. To do this, you use the m_python *order()* function. The order() function is one of the "crown jewels" of a GT.M database, allowing, with incredible simplicity, access to any of the data you store in your globals. It's not functionality that will be intuitive to a "traditional" database programmer, but it's worth understanding because it is so powerful and yet so simple.

Essentially the *order()* function is like a *forEach()* function that you'll find in languages such as PHP. The *order()* function operates on a single subscript level within a global, and returns the next subscript value in collating sequence that exists at that level in the global. You can specify some starting (seed) value, and the order() function will find the next value that exists in collating sequence. To find the first subscript at the specified level, use a starting value of null (`""`). To find the first subscript at the specified level that starts with `"C"`, use a starting value that collates just before `"C"`, for example `"B~"`

So, to find the first company in our Employee global :

```
company = ""
company = m_python.m_order(0,"^Employee",company)
```

or, more simply:

```
company = m_python.m_order(0,"^Employee","")
```

The variable c*ompany* now contains the value of the company subscript in the first record in the ^Employee global.

When the last value is found, the next time the order() function is invoked, it will return a null value. So, for example, if there was only one company in the global, if we repeat that command, ie:

```
company = m_python.m_order(0,"^Employee",company)
```

then the variable *company* will now contain a null value (ie `""`)

To get and process all companies in the Employee global, we would create a loop:

```
company = ""
while (m_python.m_order(0,"^Employee",company) <> ""):
  company = m_python.m_order(0,"^Employee",company)
  # now do something with the company you've found
```

So, we seed the order() function with a null value to make sure that it starts by finding the first subscript value that is stored in the global. We loop through each value until we exhaust the values that are saved, in which case we get a null value returned by the order() function. We then detect this null value and terminate the loop.

You'll find that this kind of loop is one of the most common things you'll do with GT.M globals.

So how would we traverse our entire ^Employee global, since it has 4 subscripts? Well, w can simply extend the logic to traverse each level. To do this we loop through each subscript level, starting at the first, and progressively moving to the next one, eg:

```
cy = ""
while (m_python.m_order(0,"^Employee",cy) <> ""):
  cy = m_python.m_order(0,"^Employee",cy)
  cntry = ""
  while (m_python.m_order(0,"^Employee",cy,cntry) <> ""):
    cntry = m_python.m_order(0,"^Employee",cy,cntry)
    ofc = ""
    while (m_python.m_order(0,"^Employee",cy,cntry,ofc) <> ""):
      ofc = m_python.m_order(0,"^Employee",cy,cntry,ofc)
      eno = ""
      while (m_python.m_order(0,"^Employee",cy,cntry,ofc,eno) <> ""):
        eno = m_python.m_order(0,"^Employee",cy,cntry,ofc,eno)
        record = m_python.m_get(0,"^Employee",cy,cntry,ofc,eno)
        # now do something with the record
```

Note here the way we've nested the *order()* functions to create a hierarchy of nested loops. Also note how the *order()* function is being used to ultimately provide values for the four subscripts of the global, so that at the inside of the loop hierarchy, we can process each and every record on file.

If we wanted to find and process only those employees in companies starting with "C", then one common method is as follows:

```
cy = "B~"
while (m_python.m_order(0,"^Employee",cy)[0] == "C"):
  cy = m_python.m_order(0,"^Employee",cy)
  #do something with the company
```

So read this loop as:

- Seed the order() function with a value that just preceeds "C" in collating sequence
- Set up a loop to find all company records in collating sequence
- If the first character of the company is a "C", then process the company record, otherwise cease traversing the global.

This ability to start and stop traversing or parsing a global at any place in the collating sequence, and at any level in the hierarchy of subscripts is pretty much unique to GT.M's Globals.

Checking to see if a Global node exists

You'll often want to know if a particular global node exists. You can use the m_apache *data()* function for this, eg:

```
data = m_python.m_data(0,"^Employee",company)
```

The data() function will actually return a number of different values.

- If data exists at the specified level of subscripting, and there are no sub-nodes, a value of '1' is returned
- If data exists at the specified level of subscripting, and there are sub-nodes, a value of '11' is returned
- If no data exists at the specified level of subscripting, but there are sub-nodes, a value of '10' is returned
- If no data and no subnodes exist at the specified level of subscripting, a value of '0' is returned.

For example, consider the following global:

```
^test=3
^test("a")=1
^test("a","b","c")=2
^test("a","b","d")=3


m_python.m_data(0,"^test")= '11'
m_python.m_data(0,"^test","a","b")= '10'
m_python.m_data(0,"^test","a","b","c")= '1'
m_python.m_data(0,"^test","x")= '0'
```

Mapping between Dictionaries and Globals

Both GT.M Globals and Python Dictionaries allow a hierarchy of name/value pairs to be represented. The key difference is that whilst a Dictionary exists in memory, a Global is stored on disk.

Mapping between Globals and Dictionaries is pretty straightforward, and if you look in the gtm.py module that is included in the M/DB Appliance, you'll see a set of fully worked examples.

You can try them out for yourself as follows. First import the gtm.py module and then create the demo dictionary:

```
ewd@mdb:/usr/local/gtm/ewd$ python
Python 2.5.1 (r251:54863, Mar  7 2008, 04:10:12)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import gtm
>>> mydict = gtm.createDictionaryExample()
>>> mydict
{'a': 111, 'c': 'xxxxx', 'b': 'hello', 'd': {'q': 'qqqq', 'r': {'x': 222, 'z': 123}, 'e': 'eeeee', 'w': 'wwwwww'}, 'arr': ['hhjg', 'lkjklj', 'uyi'], 'arr2': [{'k': 'ssss', 'l': 'hello'}, {'k': 'again', 'l': 'there'}]}
>>>
```

Now map it to a global:

```
>>> gtm.saveDictionary(mydict,"^myGlobal")
>>>
```

Now, in a separate process, let's examine the global we created from within the GT.M shell:

```
GTM>zwr ^myGlobal
^myGlobal("a")=111
^myGlobal("arr",1)="hhjg"
^myGlobal("arr",2)="lkjklj"
^myGlobal("arr",3)="uyi"
^myGlobal("arr2",1,"k")="ssss"
^myGlobal("arr2",1,"l")="hello"
^myGlobal("arr2",2,"k")="again"
^myGlobal("arr2",2,"l")="there"
^myGlobal("b")="hello"
^myGlobal("c")="xxxxx"
^myGlobal("d","e")="eeeee"
^myGlobal("d","q")="qqqq"
^myGlobal("d","r","x")=222
^myGlobal("d","r","z")=123
^myGlobal("d","w")="wwwwww"

GTM>
```

So what we've essentially done is to save the dictionary into persistent storage.

Now let's get it back:

```
>>> myDict2=gtm.getDictionary("^myGlobal")
>>> myDict2
{'a': '111', 'c': 'xxxxx', 'b': 'hello', 'd': {'q': 'qqqq', 'r': {'x': '222', 'z': '123'}, 'e': 'eeeee', 'w':
'wwwwww'}, 'arr': {'1': 'hhjg', '3': 'uyi', '2': 'lkjklj'}, 'arr2': {'1': {'k': 'ssss', 'l': 'hello'}, '2': {'k':
'again', 'l': 'there'}}}
>>>
```

Now you'll notice a slight difference from the original dictionary: the records under the "arr" and "arr2" nodes are returned as a dictionary rather than an array. The logic of the getDictionary() function could, no doubt, be modified to do this.

The reader is encouraged to examine the Dictionary-related code in the gtm.py module and improve and enhance it. I'd be quite happy to receive improvements!

The key message to you, the reader, is that GT.M can be used as a very efficient persistence engine for Python Dictionaries.

Examining Globals

GT.M includes a number of native language-based utilities for examining globals. The simplest, command line utility is the **zwr** command that was saw earlier.

```
GTM>zwr ^Employee
^Employee("C1","UK","London",1)="Steve Jones`Director`01234 9991234"
^Employee("MGW","UK","London",1)="Rob Tweed`Director`020 8404 1234"
^Employee("MGW","UK","London",2)="Chris Munt`Director`01737 2341234"

GTM>
```

You can apply filtering by using the * wildcard, eg:

```
GTM>zwr ^Employee("MGW",*)
^Employee("MGW","UK","London",1)="Rob Tweed`Director`020 8404 1234"
^Employee("MGW","UK","London",2)="Chris Munt`Director`01737 2341234"

GTM>
```

Consult the GT.M programming guide for more details about this and other utilities for examining globals.

M/DB: GT.M in Action

If you want to see examples of GT.M globals in actions, you can take a look at how M/DB uses globals to mimic the data storage behaviour of SimpleDB. Since you've already been using the M/DB Appliance for this tutorial, you have this example application already available and ready for use.

You can use the standard Amazon-recommended Python client for SimpleDB, ***boto*** (http://developer.amazonwebservices.com/connect/entry.jspa?externalID=827&categoryID=148) to access M/DB. Simply change the parameters on the *connect_sbd()* function call to:

```
connect_sdb(path="/mdb/request.mgwsi", is_secure=False, host=MDB_HOST)
```

where MDB_HOST is the IP Address/domain name assigned to your M/DB appliance machine.

Let's create a couple of M/DB records and watch how the ^MDB global (which is where M/DB stores its data) changes. In this example, M/DB's authorization has been set up for a user key of ***rob*** and a secret key of ***1234567*** and the IP address assigned to the M/DB appliance is 192.168.1.104.

In your Python shell (in /usr/local/gtm/ewd) type:

```
>>> import boto
>>> mdb=boto.connect_sdb('rob','1234567',
      path="/mdb/request.mgwsi", is_secure=False,
      host="192.168.1.104")
>>> domain = mdb.create_domain('PythonTest')
```

Now, in a separate process, take a look at the ^MDB global using the GT.M shell:

```
GTM>zwr ^MDB

^MDB("rob")=1
^MDB("rob","domainIndex","PythonTest",1)=""
^MDB("rob","domains")=1
^MDB("rob","domains",1,"created")="61688,37486"
^MDB("rob","domains",1,"modified")="61688,37486"
^MDB("rob","domains",1,"name")="PythonTest"
```

Now let's add an item:

```
>>> item = domain.new_item('item1')
>>> item['key1'] = 'value1'

Note: if you get an M/DB error returned, try entering this
command again and it should work on the second attempt.

>>> item['key2'] = 'value2'
>>> item.save()
```

Now check the ^MDB global:

```
GTM>zwr ^MDB

^MDB("rob")=1
^MDB("rob","domainIndex","PythonTest",1)=""
^MDB("rob","domains")=1
^MDB("rob","domains",1,"attribs")=2
^MDB("rob","domains",1,"attribs",0)="itemName()"
^MDB("rob","domains",1,"attribs",1)="key1"
^MDB("rob","domains",1,"attribs",2)="key2"
^MDB("rob","domains",1,"attribsIndex","itemName()",0)=""
^MDB("rob","domains",1,"attribsIndex","key1",1)=""
^MDB("rob","domains",1,"attribsIndex","key2",2)=""
^MDB("rob","domains",1,"created")="61688,38129"
^MDB("rob","domains",1,"itemIndex","item1",1)=""
^MDB("rob","domains",1,"items")=1
^MDB("rob","domains",1,"items",1)="item1"
^MDB("rob","domains",1,"items",1,"attribs",0,"value")=1
^MDB("rob","domains",1,"items",1,"attribs",0,"value",1)="item1"
^MDB("rob","domains",1,"items",1,"attribs",1,"value")=1
^MDB("rob","domains",1,"items",1,"attribs",1,"value",1)="value1"
```

```
^MDB("rob","domains",1,"items",1,"attribs",1,"valueIndex","value1
",1)=""
^MDB("rob","domains",1,"items",1,"attribs",2,"value")=1
^MDB("rob","domains",1,"items",1,"attribs",2,"value",1)="value2"
^MDB("rob","domains",1,"items",1,"attribs",2,"valueIndex","value2
",1)=""
^MDB("rob","domains",1,"modified")="61688,38180"
^MDB("rob","domains",1,"name")="PythonTest"
^MDB("rob","domains",1,"queryIndex",0,"item1",1)=""
^MDB("rob","domains",1,"queryIndex",1,"value1",1)=""
^MDB("rob","domains",1,"queryIndex",2,"value2",1)=""
```

We won't dwell here on why the ^MDB global is structured the way it is, but you can see how all the various pieces of information are being stored and indexed. In the case of M/DB, I have chosen to use just one Global (^MDB) to provide a self-contained storage container for not only the base data but all the internal indexes too. Some GT.M developers prefer to use separate Globals for indexes: it's really largely a matter of personal preference how you design your Globals.

There is no schema, per se, in M/DB. It is the application logic that maintains the schema implicitly.

The various contents of the ^MDB global are traversed using the order() function when it is queried. This is what happens behind the scenes when we run the following:

```
>>> query="SELECT * FROM PythonTest"
>>> results = mdb.select("PythonTest",query)
>>> print results
[{u'key2': u'value2', u'key1': u'value1'}]
```

The M/DB logic has been written using the GT.M native language rather than Python (though it would be entirely practical to rewrite it using Python). If you're interested in seeing the full working internals of M/DB, take a look at the file ***MDB.m*** that you'll find in the */usr/local/gtm/ewd* directory.

For further documentation on ***boto***'s SimpleDB APIs, see
http://boto.s3.amazonaws.com/ref/sdb.html

The M/DB Appliance includes a pre-built Python module, ***mdb.py***, that contains a number of examples of how the ^MDB global can be accessed directly from Python to retrieve relevant data. You'll find this module in the /usr/local/gtm/ewd directory. Feel free to look at the code it contains and see how it uses the m_python functions to navigate the ^MDB global. Here's some examples of its use. You may want to trace the code used in ***mdb.py*** to see how these functions actually work:

```
>>> import mdb
>>> mdb.getDomainsByUser('rob')
['PythonTest']
>>> mdb.getAttributeNames('rob','PythonTest')
['key1', 'key2']
>>> mdb.getItemNames('rob','PythonTest')
['item1']
>>> mdb.getAttributeValues('rob','PythonTest','item1','key1')
['value1']
>>>
```

Conclusions

We've now covered the core basics of what globals are and how they can be created and manipulated using Python. The next chapter will look at globals from the perspective of someone familiar with a relational database.

You've probably already realised that GT.M's Globals impose very few controls or limitations over what you do. That's both a great thing – it allows you to very rapidly and flexibly design, implement and redesign your database structures – and a dangerous thing – in the wrong hands it can be a recipe for a completely uncontrolled mess. GT.M leaves the discipline entirely to you, the programmer. There are no safety nets, but on the other hand, there's almost no limit to what you can achieve or how you achieve it. You'll find that the efficiency of coding and execution is what really makes GT.M an exciting and exhilarating environment to work in.

Once you try using GT.M's globals for persistent storage, you'll probably begin to wonder why all databases couldn't work this way! Its simple, intuitive, flexible and the performance significantly outstrips any relational database.

Chapter 3

RDBMS/SQL vs. GT.M Globals

This chapter will set out the salient differences between a standard Relational Database Management Systems (RDBMS) managed through SQL and the GT.M's global-based data repository. Refer back to to Chapter 2 if you need to understand more about the basics of Globals and their manipulation.

Defining the data

Let us start with the basics – defining the data.

As an example, we shall use a simple database consisting of three tables:

1. A table of customers (CUSTOMER).
2. A table of orders (ORDER).
3. A table indicating the items making up an individual order (ITEM).

```
CUSTOMER  custNo  name  address  totalOrders
|
|
+------< ORDER  orderNo  custNo  orderDate  invoiceDate  totalValue
          |
          |
          +------< ITEM  orderNo  itemNo  price
```

Table names are shown in **bold**. The primary key of each table is shown underlined.

CUSTOMER

custNo	The Customer’s (unique) number.
name	The Customer’s name.
address	The Customer’s address.
totalOrders	The number of orders placed by the customer to date.

ORDER

<u>orderNo</u>	The order number.
custNo	The relevant customer number (a foreign key from CUSTOMER).
orderDate	The Date of the Order.
invoiceDate	The Date of the Invoice.
totalValue	The Value of the order.

ITEM

<u>orderNo</u>	The order number (corresponding key from ORDER).
<u>itemNo</u>	The item (or part) number.
price	The price of the item to the customer (including any discount).

The one-to-many relationships are shown in the diagram. Each customer can have many orders and each order can be made up of any number of items (or parts).

The number of orders for a particular customer (CUSTOMER.totalOrders) is the total number of orders placed by the customer as identified by ORDER.orderNo. The value of an order (ORDER.totalValue) is the sum of the cost of each item in the order (as identified by ITEM.price).

CUSTOMER.totalOrders and ORDER.totalValue are not directly entered by a user of the application - they are derived fields.

For an RDBMS, these table definitions must be entered into the data definition part of the database (or schema) before SQL can be used to add, modify and retrieve records.

GT.M does not enforce the use of a data-definition scheme and, as such, records can be written directly to the GT.M global-baseddata store without the need to formally defined the data. However, it is important to note that a relational schema can easily be layered on top of a set of globals for the purpose of accessing the GT.M data via SQL-based reporting tools. A relational schema can be retrospectively added to an existing GT.M data store provided the records are modeled to (approximately) third normal form.

The tables can be represented in GT.M using the following globals:

CUSTOMER

^CUSTOMER(custNo)=name|address|totalOrders

ORDER

^ORDER(orderNo)=custNo|orderDate|invoiceDate|totalValue

ITEM

^ITEM(orderNo,itemNo)=price

The relationship between the CUSTOMER and ORDER tables could be represented by another global:

^ORDERX1(custNo,orderNo)=""

This would provide pointers to all Order Numbers for a specified Customer Number.

In GT.M, it's up to you what global names you use. Also you have a choice of either using different globals for each table (as we've shown above), or using the same global for some or all the tables and indices. For example, we could pack everything together using the following global structure:

```
^OrderDatabase("customer",custNo)= name_"~"_address_"~"_totalOrders
^OrderDatabase("order",orderNo)= custNo_"~"_orderDate_"~"_invoiceDate_"~"_totalValue
^OrderDatabase("item",orderNo,itemNo)=price
^OrderDatabase("index1",custNo,orderNo)=""
```

For the purposes of the examples that follow, we've chosen to use separate globals for each table.

We've also decided, arbitrarily, to use a tilde (~) character as the data delimiter in our example globals.

Adding new records to the Database

Let's start with a really simple example. Let's add a new customer record to the customer table.

SQL

```
INSERT INTO CUSTOMER (CustNo, Name, Address)
VALUES (100, 'Chris Munt', 'Oxford')
```

GT.M/Python

```
m_python.m_set(0, "^CUSTOMER", 100, "Chris Munt~Oxford")
```

We are using the tilde character (~) as the field delimiter. We can, of course, use any character we like for this purpose (including non-printable characters). For example, we could use:

```
record = "Chris Munt" + chr(1) + "Oxford"
m_python.m_set(0, "^CUSTOMER", 100, record)
```

Of course, in a real situation, the data values would be passed to the insert query (or Python method) by means of a program as values of variables:

SQL

```
INSERT INTO CUSTOMER (custNo, :name, :address)
VALUES (:custNo, :name, :address)
```

GT.M/Python

```
record = name + "~" + address
m_python.m_set(0, "^CUSTOMER", custNo, record)
```

Retrieving records from the database

SQL

```
SELECT A.name, A.address
FROM CUSTOMER A
WHERE A.custNo = :custNo
```

GT.M/Python

```
record = m_python.m_get(0, "^CUSTOMER", custNo)
field = record.split("~")
name = field[0]
address = field[1]
```

Removing records from the database

SQL

```
DELETE FROM CUSTOMER A
WHERE A.custNo = :custNo
```

GT.M/Python

```
m_python.m_kill(0, "^CUSTOMER", custNo)
```

Note that this simple example doesn't take referential integrity into account. You'll see later how we'll deal with this.

Parsing the database

SQL

```
SELECT A.custNo, A.name, A.address
FROM CUSTOMER A
```

GT.M/Python

```
 custNo = ""
 while (m_python.m_order(0,"^CUSTOMER",custNo) <> ""):
   custNo = m_python.m_order(0,"^CUSTOMER",custNo)
   record = m_python.m_get(0, "^CUSTOMER", custNo)
   field = record.split("~")
   name = field[0]
   address = field[1]
   totalOrders = field[2]
   # add code here to process the "selected row"
```

The order() function is one of the keys to the power and flexibility of globals. Its functionality is not likely to be intuitive to someone familiar with RDBMS and SQL, so it's word reading the more detailed description in Chapter 2.

With the order() function and globals, you have total access to step through any of the keys in a table, starting and finishing on any value we wish. The important thing is to understand that Globals represent a hierarchical data storage structure. The key fields in the tables we're emulating are represented as the subscripts in a global, so we no longer have to access rows from a table in strict sequence: the order() function can be applied to each subscript (key) independently.

Using GT.M/Python functions to encapsulate database access

In practice, and in order to maximise code reuse, the GT.M queries shown previously would usually be implemented as Python functions. Examples of such functions are shown below.

Adding new records to the database:

```
def setCustomer(custNo,name,address):
  if custNo == "":
    return False
  record = name + "~" + address
  m_python.m_set(0, "^CUSTOMER", custNo, record)
  return True
```

This function could be now called repeatedly in the following way:

```
name="Rob Tweed"
address="London"
custNo=101
setCustomer(custNo,name,address)
```

Note the conditional statement that prevents use of a null value for the Customer Number. This is because Global subscripts cannot be null values. Attempting to use a null subscript value will return a GT.M error.

Because we have a derived field (totalOrders) within the CUSTOMER table, the GT.M/Python code would more properly be:

```
def setCustomer(custNo,name,address):
  if custNo == "":
    return False
  # derive the number of orders
  noOfOrders = 0
  orderNo = ""
  while (m_python.m_order(0,"^ORDERX1",custNo,orderNo) <> ""):
    orderNo = m_python.m_order(0,"^ORDERX1",custNo,orderNo)
    noOfOrders += 1
  record = name + "~" + address + "~" + noOfOrders
  m_python.m_set(0, "^CUSTOMER", custNo, record)
  return True
```

We will discuss these derived fields later in the 'triggers' section.

Retrieving records from the database:

The following Python function will retrieve a row from the CUSTOMER table, returning the data as a Dictionary.

```
def getCustomer(custNo):
  if custNo == "":
    return {}
  record = m_python.m_get(0, "^CUSTOMER", custNo)
  if record == "":
    return {}
  field = record.split("~")
  name = field[0]
  address = field[1]
  totalOrders = field[2]
  return {"name":name,"address":address,"totalOrders":totalOrders}
```

This function can be used as follows:

getCustomer(100)
{'totalOrders': '1', 'name': 'Chris Munt', 'address': 'Oxford'}

Removing records from the database:

The following extrinsic function will delete a row from the CUSTOMER table:

```
def deleteCustomer(custNo):
  if custNo == "":
    return False
  m_python.m_kill(0, "^CUSTOMER", custNo)
  return True
```

This function can be used as follows:

custNo = 101
deleteCustomer(custNo)

Secondary Indices

In an RDBMS, secondary indices are maintained within the data-definition layer. Having defined them there, the contents of the index are automatically created and maintained by the RDBMS.

Indexes within GT.M Globals are maintained explicitly, for example in the table's update function. Because of the hierarchical nature of Globals, the primary record doubles as an index based on the primary key.

For example, consider the GT.M/Python function for adding a new record to the ORDER table:

```
def setOrder(orderNo,custNo,orderDate,invoiceDate):
  if orderNo == "":
    return False
  # Calculate the value of the order
  totalValue = 0
  itemNo = ""
  while (m_python.m_order(0,"^ITEM",orderNo,itemNo) <> ""):
    itemNo = m_python.m_order(0,"^ITEM",orderNo,itemNo)
    value = getItemValue(orderNo,itemNo)
    totalValue = totalValue + value
  record = custNo + "~" + orderDate + "~" + invoiceDate
  record = record + "~" + totalValue
  m_python.m_set(0, "^ORDER", orderNo, record)
  return True
```

Notice the code for calculating the value of the order. This parses all the ITEM records for the specified Order Number, and uses the getItem() function to reteieve the value for each item. The getItemValue() function would contain the following:

```
def getItemValue(orderNo,itemNo):
   if orderNo == "":
     return 0
   if itemNo == "":
     return 0
   value = m_python.m_get(0, "^ITEM, orderNo, itemNo)
   if value == "":
     return 0
   return value
```

Now, suppose we wish to add the index to facilitate easy access to orders on a per-customer basis. We're representing this by another global named ^ORDERX1. To do this we create an index on Customer Number (custNo) and Order Number (orderNo). This would be implemented by extending the setOrder function as follows:

```
def setOrder(orderNo,custNo,orderDate,invoiceDate):
  if orderNo == "":
    return False
  # Calculate the value of the order
  totalValue = 0
  itemNo = ""
  while (m_python.m_order(0,"^ITEM",orderNo,itemNo) <> ""):
    itemNo = m_python.m_order(0,"^ITEM",orderNo,itemNo)
    value = getItemValue(orderNo,itemNo)
    totalValue = totalValue + value
  record = custNo + "~" + orderDate + "~" + invoiceDate
  record = record + "~" + totalValue
  m_python.m_set(0, "^ORDER", orderNo, record)
  if custNo <> "":
    m_python.m_set(0, "^ORDERX1", custNo, orderNo, "")
  return True
```

Referential Integrity

Referential Integrity is, perhaps, the best known of a general class of 'referential actions'. Referential actions include all operations that involve changes to tables that must be made as a direct consequence of modifications to other tables.

The process of maintaining Referential Integrity is responsible for maintaining semantic integrity between related tables. Specifically, it is concerned with maintaining the natural joins or primary/foreign key correspondences between tables.

For example, when a customer number (CUSTOMER.custNo) is changed from one value to another (or removed), in order to maintain semantic integrity between the customer table and the order table, a corresponding change is indicated in the order table (ORDER.custNo). Similarly, when an order number (ORDER.orderNo) is changed from one value to another (or removed), in order to maintain semantic integrity between the order table and the item table, a corresponding change is indicated in the item table (ITEM.orderNo).

In an RDBMS, these integrity rules are implied by the information contained within the data definition layer (primary and foreign keys). In GT.M, these rules can be implemented directly within our Python functions for maintaining the data store.

Taking the relationship between the customer and the order table, an update operation for CUSTOMER would be expressed as:

SQL

```
UPDATE CUSTOMER A
SET custNo = :newCustNo
WHERE A.custNo = :oldCustNo
```

This query will result in the corresponding records being updated in the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo)

GT.M/Python

```
def updateCustomer(oldCustNo,newCustNo,newName,newAddress):
  if oldCustNo == "":
    return False
  if newCustNo == "":
    return False
  orderNo = ""
  while (m_python.m_order(0,"^ORDERX1",oldCustNo,orderNo) <> ""):
    orderNo = m_python.m_order(0,"^ORDERX1",oldCustNo,orderNo)
    orderData = getOrder(orderNo)
    orderDate = orderData["orderDate"]
    invoiceDate = orderData["invoiceDate"]
    setOrder(orderNo,newCustNo,orderDate,invoiceDate)
  setCustomer(newCustNo,newName,newAddress)
  if oldCustNo <> newCustNo:
    deleteCustome(oldCustNo)
  return True
```

Notice that this function is largely built using Python functions that we have already created for the purpose of maintaining the customer and order table.

We'd need to create the getOrder function :

```
def getOrder(orderNo):
  if orderNo == "":
    return {}
  record = m_python.m_get(0, "^ORDER", orderNo)
  if record == "":
    return {}
  field = record.split("~")
  custNo = field[0]
  orderDate = field[1]
  invoiceDate = field[2]
  totalValue = field[3]
  return {"custNo":custNo,"orderDate":orderDate,
   "invoiceDate":invoiceDate,"totalValue":totalValue}
```

We'll also need to extend our original, simple deleteCustomer() function. Similar considerations apply to operations that remove customer records from the database. The SQL query together with its equivalent M function is shown below.

SQL

```
DELETE FROM CUSTOMER A
WHERE A.custNo = :custNo
```

This query will result in the corresponding records being deleted from the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo and the integrity rules specified within the data-definition layer).

GT.M/Python

```
def deleteCustomer(custNo):
  if custNo == "":
    return False
  orderNo = ""
  while (m_python.m_order(0,"^ORDERX1",oldCustNo,orderNo) <> ""):
    orderNo = m_python.m_order(0,"^ORDERX1",oldCustNo,orderNo)
    result=deleteOrder(custNo,orderNo)
  m_python.m_kill(0,"^CUSTOMER",custNo)
  return True
```

This requires the deleteOrder() function:

```
def deleteOrder(custNo,orderNo):
  m_python.m_kill(0,"^ITEM",orderNo)
  m_python.m_kill(0,"^ORDER",orderNo)
  m_python.m_kill(0,"^ORDERX1",custNo,orderNo)
  return True
```

Note that all item records in the ITEM table can be removed by applying the kill command at the orderNo hierarchy level.

This assumes that a ‘cascading’ relationship is required between the customer and order table. A ‘breaking link’ relationship would be implemented as follows:

```
def deleteCustomer(custNo):
  if custNo == "":
    return False
  orderNo = ""
  while (m_python.m_order(0,"^ORDERX1",custNo,orderNo) <> ""):
    orderNo = m_python.m_order(0,"^ORDERX1",custNo,orderNo)
    orderData = getOrder(orderNo)
    orderDate = orderData["orderDate"]
    invoiceDate = orderData["invoiceDate"]
    setOrder(orderNo,custNo,orderDate,invoiceDate)
  m_python.m_kill(0,"^CUSTOMER",custNo)
  return True
```

Similar logic will apply with respect to data held in the ITEM table when an Order Number (ORDER.orderNo) is modified in, or deleted from the ORDER table.

Triggers

Triggers are simply a way of automatically invoking pre-defined code when certain conditions are met within the database. A trigger is usually fired as a result of records being updated in a certain way within the database.

In an RDBMS, triggers are defined within the data definition layer. In GT.M we can add trigger code directly to the Python functions that we write for the purpose of maintaining the data store.

For example, consider the number of orders for a customer (CUSTOMER.totalOrders). We could define a trigger to automatically update this field as records are added (or removed) from the order table.

SQL

The following SQL would be included in the trigger code within ORDER for the purpose of creating CUSTOMER.totalOrders:

```
SELECT COUNT(A.orderNo)
FROM A.ORDER
WHERE A.custNo = :custNo
```

This query will be triggered for all constructive and destructive operations that are applied to the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo)

GT.M/Python

We simply add the following (trigger) code to the function for adding records to the order table:

```
def setOrder(orderNo,custNo,orderDate,invoiceDate):
  if orderNo == "":
    return False
  # Calculate the value of the order
  totalValue = 0
  itemNo = ""
  while (m_python.m_order(0,"^ITEM",orderNo,itemNo) <> ""):
    itemNo = m_python.m_order(0,"^ITEM",orderNo,itemNo)
    value = getItemValue(orderNo,itemNo)
    totalValue = totalValue + value
  record = custNo + "~" + orderDate + "~" + invoiceDate
  record = record + "~" + totalValue
  m_python.m_set(0, "^ORDER", orderNo, record)
  if custNo <> "":
    m_python.m_set(0, "^ORDERX1", custNo, orderNo, "")
  # Trigger the update of the CUSTOMER.totalOrders field
  custData = getCustomer(custNo)
  ok = setCustomer(custNo,custData["name"],custData["address"])
  return True
```

The same considerations apply to operations that remove data from the ORDER table.

A similar scheme can be employed for the purpose of automatically updating the value of an order (ORDER.totalValue) as items are added to the order.

SQL

The following SQL would be included in the trigger code for the ITEM table for the purpose of generating ORDER.Value:

```
SELECT SUM(A.price)
FROM A.ITEM
WHERE A.orderNo = :orderNo
```

This query will be triggered for all constructive and destructive operations that are applied to the ITEM table (according to the join ORDER.orderNo = ITEM.orderNo)

GT.M/Python

We simply add the following (trigger) code to the Python function for adding records to the ITEM table:

```
def setItem(custNo,orderNo,itemNo,price)
  if orderNo == "":
    return False
  if itemNo == "":
    return False
  m_python.m_set(0, "^ITEM", orderNo, itemNo, price)
  m_python.m_set(0, "^ORDERX1", custNo, orderNo, "")
  # Trigger the update of the ORDER.totalValue field
  orderData = getOrder(orderNo)
  orderDate = orderData["orderDate"]
  invoiceDate = orderData["invoiceDate"]
  ok = setOrder(orderNo,custNo,orderDate,invoiceDate)
  return True
```

The same considerations apply to operations that remove data from the ITEM table.

A further example of the use of triggers in our customer and orders database would be to automatically generate and raise an invoice for the customer as soon as an invoice date is added to the order table (ORDER.invoiceDate). We can very simply add this functionality to our procedure for updating the ORDER table:

```
def setOrder(orderNo,custNo,orderDate,invoiceDate):
  if orderNo == "":
    return False
  # Calculate the value of the order
  totalValue = 0
  itemNo = ""
  while (m_python.m_order(0,"^ITEM",orderNo,itemNo) <> ""):
    itemNo = m_python.m_order(0,"^ITEM",orderNo,itemNo)
    value = getItemValue(orderNo,itemNo)
    totalValue = totalValue + value
  record = custNo + "~" + orderDate + "~" + invoiceDate
  record = record + "~" + totalValue
  m_python.m_set(0, "^ORDER", orderNo, record)
  if custNo <> "":
    m_python.m_set(0, "^ORDERX1", custNo, orderNo, "")
  # Trigger the update of the CUSTOMER.totalOrders field
  custData = getCustomer(custNo)
  ok = setCustomer(custNo,custData["name"],custData["address"])
  # Raise an invoice if an invoice date is entered
  if invoiceDate <> "" result=invoiceOrder(orderNo)
  return True
```

Of course, the Python function invoiceOrder() would have to be written to carry out the appropriate processing logic to raise the invoice.

Transaction Processing

A complete update to a database often consists of many individual updates to a number of tables. All contributing updates must be guaranteed to complete before the whole update (or *transaction*) can be said to be complete.

In an RDBMS, transaction control is usually active as a matter of default behaviour. In other words, changes to the database are not committed until the modifying process issues a ‘commit’ command, at which point the updates are committed to the database and a new transaction begins.

GT.M is very similar in this respect with the exception that transaction control has to be explicitly turned-on. This is currently only available if you define the transaction logic using GT.M's native language code: a program must issue a ‘transaction start’ command in addition to the ‘transaction’ commit command.

SQL

Commit a transaction and (implicitly) start a new transaction:

```
COMMIT
```

GT.M

Start a new transaction:

```
TSTART
```

Commit a transaction:

```
TCOMMIT
```

If transaction control is *not* explicitly switched on in a GT.M system, then all updates for that process will be immediately made to the live data store. In other words, each and every SET or KILL of a global node will be regarded as a complete transaction in its own right.

A Python programmer using the m_python gateway cannot directly specify transactions, but the *m_proc()* function could be used to invoke transaction logic that is written in GT.M's native language.

Conclusions

There are many key advantages to using GT.M over a traditional RDBMS.

- Maintainability and core reuse. With discipline, an extremely high level of code reuse can be achieved. The examples we’ve described in this paper demonstrate how you can encapsulate all your database actions in terms of *get*, *set* and *delete* functions for each table – write these once and re-use them.

- Transportability. The definition of the data is contained within the functions that operate on it. There is no division between definition and implementation.

- Flexibility. The update code can be modified to trigger any action or event within the system.

- Performance and fine-tuning. Experienced GT.M analysts will recognise opportunities for tuning the functions shown here for maintaining the data store. This is possible because the definition of the data and the implementation of the operations that are applied to it are held together within discrete functions. The analyst using SQL has no fine control over the way triggers, referential actions or transactions are managed.

- The advantages of using SQL to maintain or retrieve the data (use of third party reporting tools, for example) are not lost because a relational data definition can be transparently layered over the top of an existing set of GT.M tables (globals). This can even be done retrospectively. For example, the third-party KB_SQL product (http://www.kbsystems.com/), implements a full SQL environment, layered on GT.M's globals.