**DRAFT, February 22, 2002**

# MetaKit: Quick and Easy Storage for your Tcl Application[1]

*Mark Roseman*

*mark@markroseman.com*

Ever needed a simple and powerful way to store and retrieve data in your latest Tcl masterpiece?

Finding yourself cursing at the explosion of small data files littering your disk as your application grew?

Would you like to get a handle on efficiently extracting the right results from your data, without having to digest a 5-pound book on SQL to do it?

Need storage that won't require a painful database installation, a dedicated database analyst to maintain it, and all the associated Tcl extensions needed to get it to work with your code?

If you're like most developers, you probably would have answered "YES!!!" to these questions on several different occasions. If so, you could probably benefit from MetaKit, a storage engine that's ideally matched to Tcl: dynamic, flexible, easy to learn, embeddable, and very powerful.

In this document, we'll explore how you can use MetaKit to solve the data storage needs of your Tcl applications. You'll find out what makes MetaKit different than other approaches, and whether it's the right approach for you. You'll see how to use MetaKit from within your own Tcl programs, and learn the small number of MetaKit commands that you'll need to use for most applications.

In no time, you'll be proficient enough that you'll never (well, ok, hardly ever) look at a flat file storage solution again. So lets start by taking a closer look at what MetaKit provides you with.

---

# What is MetaKit?

MetaKit is a freely available, cross platform, open source (MIT-style licensed) database library that you can include in your own programs. It has been developed by Jean-Claude Wippler, the man behind Equi4 Software (www.equi4.com). MetaKit comes with an API for C++, one for Python, and the one we're most interested in here, an API for Tcl called "Mk4tcl".

Like every other database or storage library, MetaKit helps you manage the data you want to store on disk. However, it does things somewhat differently from many other database libraries, which give it a number of unique advantages, some of which are ideally paired to scripting languages like Tcl. We'll explore some of these differences here.

## A First Example

But first, lets look at a quick example to give you a feel for what a Tcl program using MetaKit might look like. This admittedly contrived example shows how you can store information into a data file and then easily search for particular pieces of that information. If you already have MetaKit installed on your system (which is done exactly like any other Tcl package), you can actually try this one out right now. If not, we'll provide details at the end of this document on how to obtain and install MetaKit.

```
# load MetaKit into our application; Mk4tcl is the name of the Tcl extension
package require Mk4tcl

# open a datafile named mydata.db; we'll refer to it with the tag 'db'
mk::file open db mydata.db

# create a view within the datafile which describes what we'll store
set view [mk::view layout db.addressbook "name country"]

# create a bunch of new rows in the view to store our data
mk::row append $view name "Mark Roseman" country "Canada"
mk::row append $view name "Jean-Claude Wippler" country "The Netherlands"
mk::row append $view name "Jeff Hobbs" country "Canada"
mk::file commit $view

# search for all living in Canada and print their names
foreach row [mk::select $view country "Canada"] {
    puts [mk::get $view!$row name]
}

# close the datafile
mk::file close db
```

So in a few lines of not-very-complex code, we can open a database (creating it if needed), say what we want to store in it (which may even be slightly different than the last time we opened it; MetaKit will adjust as needed), write some data, and do a simple search. That's all that's needed; we didn't need to install or set up a special database daemon somewhere on our system, etc. Everything that's needed is right in the MetaKit Tcl extension itself.

## Easy to Learn

As the simple program above suggests, one of MetaKit's strong points is that it's very easy to learn. While a bit of familiarity with "standard" database concepts will go a long way, you certainly don't need to be an expert to be proficient with MetaKit. In particular, you certainly don't need to learn the nuances of complex query languages such as SQL. As with many things in Tcl, the level of abstraction is kept high enough so that your projects don't need a large team, where each member is proficient in a single area. Instead, a single person can handle multiple areas of the program, and with MetaKit, that includes data storage.

MetaKit's API relies on concepts familiar to programmers, where data is accessed by indexing into a set of rows, each row containing a number of named properties. By relying on a straightforward and very concrete model, programmers don't get hung up learning more abstract concepts right at the start.

## Scriptable and Dynamic

Part of what contributes to MetaKit's ease of learning and ease of use is how naturally it fits into the scripting language paradigm. It's very natural to open up a Tcl shell and start typing in MetaKit commands. Everything is open to very easy inspection, such as the structure of your data file, all the data contained in it, "cursors" showing you your position as you iterate through rows in the file, and so on.

MetaKit's not only lets you inspect data files, but even change their structure on the fly. If in your application, you decide to add a new field to the database, you can simply add the new field to the "view" and the data file will automatically be adjusted. Unlike with a lot of databases, this gives you a lot of flexibility even in the early prototype stages of your application.

## Easily Embedded and Deployed

One of MetaKit's exceptional advantages is how easily your applications and databases can be deployed. We've already touched on the fact that it doesn't require a separate database application to be installed on your system, but just the MetaKit library, linked in or dynamically loaded into your application. The library itself is quite small, only several hundred kilobytes in size.

The data files you store are themselves self-contained, i.e. you needn't have to worry about multiple files containing your database. For those of you deploying applications that run on multiple platforms, MetaKit data files have the additional advantage of being easily moved between platforms, with no conversions required. And MetaKit is available for a variety of platforms, including Windows, Mac, Linux, Solaris and a variety of other Unix systems.

## Reliable and Efficient

A critical requirement of any database or storage library, MetaKit goes to great pains to ensure that your data is always safely stored. Your data files will not become corrupted.

MetaKit is also very efficient. By using internal implementation techniques like "column-wise storage" and memory mapped files (both of which map nicely onto the API), data access and manipulation is almost always extremely fast.

## What MetaKit is Not

Of course, given some of these advantages, there are tradeoffs. There are some things that MetaKit does not provide, that may or may not be important to your particular application.

*SQL*. MetaKit has its own API, and does not conform to the SQL API used by many higher end database products. If you are migrating code from an existing SQL database, or require that you be able to migrate to a SQL database, MetaKit may not be the right choice for you. That being said, SQL implementations vary considerably between products regardless, so any migration is bound to involve some effort. Similarly, if you require extremely complex queries or sophisticated data constraints, which SQL was designed to deal with, MetaKit may not be a good fit.

*Concurrent*. Only a single process (or thread) at a time can access a MetaKit data file, unlike many other products that allow multiple processes or threads to access the database at once. This is common in web servers for example, where different CGI processes will share access to the same database. Underneath of course, the database itself mediates access to the data. With MetaKit, your application would have to mediate access. A common architecture for addressing this problem is to have a single server process that accesses the database, with clients (such as the CGI processes in this example) making database requests of this server. We won't cover this here.

*Massively Scalable*. MetaKit will comfortably handle data files up to several hundred megabytes, and with a few tricks, you can push things a fair bit further. However, if you're looking to store data files of several gigabytes or more, you will probably need to look to alternative solutions.

# MetaKit Concepts

There are only a few concepts that you'll need to know to use MetaKit. Most of these should be familiar to you if you've used any other database or storage mechanism, though perhaps under a different name.

*Datafile*. MetaKit stores all its data in one or more self-contained data files on disk. These live like any other file in the file system; you open them with MetaKit when you want to access your data, and you close them afterwards. When you open a file, you specify a 'tag' that is associated with the open file (e.g. "db" in the earlier example). You can have multiple data files open at once, where each would have a different tag.

*View*. Views let you partition your data files into one or more separate areas, each of which may hold different types of data. The description of what data each view can hold is referred to as its layout, or structure. Views are specified as tag.viewname (e.g. "db.addressbook" earlier). Views are equivalent to 'tables' in many other databases.

*Row*. A row holds a collection of data related to the same object, such as the name, address, etc. of a person in an address book. This is commonly referred to as a 'record' in many other databases. A view is actually made up of an array of rows, which are referred to by a zero-based index (i.e. the first row is '0', the second is '1', etc.). You can refer to an individual row within a view by tag.viewname!index (e.g. "db.addressbook!3").

*Properties*. A property is an individual data item. Each row will hold one or more properties. Each row within a single view will contain the same properties, though of course they will likely have different values. So for example, every row within the view may have a name and address property, but the values will be different for each row. Rows are commonly known as 'fields' in many other databases. Note that properties are not referred to directly by a notation like for views and rows, but accessed through MetaKit commands like mk::get and mk::set.

# Managing Data Files

The example earlier actually showed you all you normally need to worry about data files.

Pick a file on disk to store your data. Use the "`mk::file open` *`tag filename`*" command to open the file and associate a tag with that file, e.g. "`mk::file open db mydata.db`" to create or open the mydata.db file and associate the tag "db" with the file.

As you make changes, you should periodically call "`mk::file commit` *`tag`*" to flush your changes to disk.

When you're done with the datafile, simply call "`mk::file close` *`tag`*".

That's all there is to that.

# Structuring What You Store

After opening your data files, the first thing you'll need to do is specify what kind of data is stored in them. Remember that your data files are structured as one or more *views*, where each view acts like a big array to hold your data in a set of rows. Because each row in a single view must have the same set of properties —the same structure—you'll have to define a view for each type of data you want to store in your data file.

Different types of data will require different properties. For example, an "employee" in a Human Resources application will hopefully have different properties (e.g. name, salary, etc.) than a "city" in a geographic application (e.g. latitude, longitude, name, population). Even if two data types did have the same structure (e.g. "name" and

"id" might apply to a lot of things) you still wouldn't want to confuse them, so you would still want to create two separate views.

To specify a view for a certain type of data, first identify all the properties that you would want to store for objects of that type. Make sure to break things up into as small pieces as make sense for your application, rather than combining several items into a single property. For example, many applications would be better served by multiple properties like "street address", "city", "state" and "zip code" rather than a single "address" which combines all four individual pieces. Combining them into one would for example make it harder to search for all entries from a particular country.

Once you've listed out all the properties, you have to give each a name that you'll use to refer to the property in your program. This will be like a variable in your program; so use things like "employeeid" rather than "Employee Identification Number". As with naming any variables, try to be specific and unambiguous. If you have several different types of data each referred to by a different unique identifier, use the more explicit "employeeid" and "partid" rather than a simple "id".

## Defining the View

To actually create the view and give it its structure, use the "`mk::view layout`" command. This takes a view name (a data file tag plus the name you'd like to assign to this view) and a list of properties as arguments. For example:

```
mk::view layout db.employees "name employeeid salary"
mk::view layout db.cities "cityname latitude longitude country population"
```

You have to issue this command to create the view after first creating the data file (via `mk::file open`) and before storing any data in it. You should also issue this command after opening your existing data file (again, via `mk::file open`). Since in MetaKit there's no explicit difference between creating a new file and opening an old one, it's best to always specify the view immediately after opening the file.

# Storing and Retrieving Data

Now that we've defined what we're going to store in a view, it's time to see how to actually store something there. In this section, we'll assume we're dealing with employee records, using a view set up as follows:

```
mk::file open db mydata.db
mk::view layout db.employees "name employeeid salary"
```

## Adding Rows

Remember that each view holds an array of rows, indexed by position (zero being the first, one being the second, etc.). Initially, a view will have no rows stored in it. The obvious thing then to do is add a new record, which can be done via the "mk::row append" command, as follows:

```
set row [mk::row append db.employees]
```

This will add a new row at the end of any existing rows. Initially, all the properties for this row will be set to empty strings. The command will return a string that you can use to refer to the individual row. For the first row created, this will be "`db.employees!0`", for the second "`db.employees!1`", and so on. You can also set one or more properties for the new row by passing the name and values of the properties to the `mk::row append` command:

```
mk::row append db.employees name "Mark" employeeid 1 salary 10
mk::file commit db
```

The command "`mk::view size` *`viewname`*" command will tell you how many rows are in the specified view.

## Setting Properties

To set one or more properties for a row, use the "mk::set" command, which takes a row reference (i.e. `db.viewname!rownum`) and a list of one or more property names and values. So for example to change the salary of the employee at index 3 in the employees view, we could use:

```
mk::set db.employees!3 salary 25
```

The last `mk::row append` in the previous section could also be written as:

```
set row [mk::row append db.employees]
mk::set $row name "Mark" employeeid 1 salary 10
```

Remember to use "`mk::file commit`" periodically after storing data in your files.

## Retrieving Properties

The `mk::get` command is used to retrieve the values of one or more properties for a row. There are two forms. In the first, you provide a list of properties you'd like to retrieve, and the command returns a list of the values of those properties. For example:

```
puts [mk::get db.employees!0 name employeeid]
→ Mark 1
```

In the second form, when mk::get is called without a list of property names, the command will return a list of all properties and their values:

```
puts [mk::get db.employees!0]
→ name Mark employeeid 1 salary 10
```

## Iterating over Rows

If you just want to iterate over all the rows in a view, for example to do simple reporting or batch modifications, you can do this with the "`mk::loop`" command. Like "for" loop constructs in programming languages, a loop variable will be set to point to each row in turn. For example, to print out the names of all employees in our example:

```
mk::loop i db.employees {
    puts [mk::get $i name]
}
```

## Deleting Rows

You can delete one or more rows using the "mk::row delete" command. For example:

```
mk::row delete db.employees!2    # delete a single row
mk::row delete db.employees!5 3  # delete three rows starting at index 5
```

In general, you can't use "mk::row delete" within a "mk::loop" construct. Because "mk::loop" uses the index (row number) to keep track of the current item, and "mk::row delete" removes rows, the loop counter will get thrown off.

# Queries

If you're searching for only particular data items, its possible to do it simply by iterating over each row and comparing the row's properties to what you're looking for. However, its easier and more efficient to use the "`mk::select`" command, which will do the work of searching through the rows for you.

The `mk::select` command will search through the rows in a view and return a list of row numbers matching a given search criterion. So to search for all employees with a salary of 10, we could do the search, and iterate over these rows to print their names:

```
set rownums [mk::select db.employees salary 10]
foreach i $rownums {
    puts [mk::get db.employees!$i name]
}
```

The search criterion is specified as a property followed by its value. To search for rows matching multiple criterion you can specify multiple property/value pairs (e.g. "salary 10 name Mark"), where all of the properties must match. The default matching is case-insensitive.

## Other Options

You can also specify other search options, beyond just a simple check if a property equals a given value ("`property value`"). These include:

| | |
|---|---|
| -min *property value* | Property must be greater than or equal to value. |
| -max *property value* | Property must be less than or equal to value. |
| -exact *property value* | Exact case-sensitive string match. |
| -glob *property value* | Glob-style (i.e. string match) matching. |
| -globnc *property value* | Case-insensitive glob-style matching. |
| -regexp *property value* | Match by regular expression. |
| -keyword *property value* | Match if value is included as a word or prefix of a word in the property. |

You can also sort the results according to the value of a property ("`-sort` *`property`*" or reverse ordered sort with "`-rsort` *`property`*"). There are also other options allowing you to choose the maximum number of results to return, at what index to start the search, and so on. See the reference manual for details.

# Common Idioms

So far we've covered all the MetaKit commands you'll normally need for your application, including working with data files, specifying views, adding, modifying and deleting rows, and searching for particular rows.

In this section, we'll describe some common programming idioms that you'll find yourself using with your MetaKit applications. These are just some simple techniques that you'll find make building your programs easier.

## Multiple Views

This one is simple; remember that if you're storing different types of data, you should place them in different views. Using multiple "`mk::view layout`" statements, you can effectively partition a single data file into multiple different areas. Don't try to take shortcuts by playing tricks like "if this property is set to this value, its really this type of data, so this other field means…" You'll only get burned later. Set up a separate view.

## Generating Unique Ids

In MetaKit, you've seen that individual rows are referred to by their row number (i.e. their position or index within the entire view). Because this can change (when rows are inserted or deleted), it's often valuable to assign a unique identifier to the object, to refer to it over a longer-term basis. For example, employees may be assigned a unique employee id, etc. (such ids are often referred to as the "primary key").

One approach to generating these unique ids in MetaKit is to use a separate "control" view within the data file. This view would hold a single row, containing a counter used to generate this id. For example:

```
mk::file open db mydata.db

# create a control view holding our counter
mk::view layout db.control {
    nextid:I
}

# initialize it if needed
if {[mk::view size db.control]==0} {
    mk::row append db.control nextid 1
}

#  procedure to return a unique identifier
proc getuniqueid {} {
    set id [mk::get db.control!0 nextid]
    mk::set db.control!0 nextid [expr $id+1]
    mk::file commit db
    return $id
}
```

## Reading/Writing from Tcl Arrays

The "`mk::get`" command returns a Tcl list in the format of "*`property value property value`…". We can take advantage of the fact that this is the same format used by the Tcl "`array set`" command. This allows you to store the contents of an entire row into a Tcl array, and then use commands like "`info exists`", "`array names`", etc. to examine the properties, as well as constructs like "`$`*`arrayname`*`(`*`property`*`)`" to refer to a property.

Similarly, after you've read a row into an array and made some changes (or constructed the array manually), you can write it back using a construct like "`eval mk::set `*`db.viewname`*`!`*`rownum`* `[array get `*`arrayname`*`]`".

# More Advanced Techniques

This section will cover a few more advanced features of MetaKit that you'll probably find yourself soon including in your programs, after you've mastered the basics.

## Data Types

Normally, every property you store in a view will be stored as a string. In MetaKit, you don't have to predefine the lengths of the strings you'll store; the data file will adjust to accommodate whatever length of strings you store.

If you want, you can specify that some properties are stored not as strings, but as other data types. This can be handy because it helps MetaKit store things more efficiently. If all values assigned to a property are fairly small integers, telling MetaKit that only integers will be stored in that property makes for more efficient storage. As with strings, MetaKit will attempt to use the smallest amount of storage possible. If you specify that a field will hold integers, and you only ever store values of 0 or 1, MetaKit will pack this integer property for eight different rows into a single byte.

Data types are specified by adding a "type specifier" to the name of the property when you specify it in the `mk::view layout` command. You can't have two properties with the same name but different type specifiers. For example, the type specifier for an integer property is ":I", so you might use something like:

```
mk::view layout db.employees "name employeeid:I salary"
```

Here is the list of valid data types and the corresponding type specifiers:

| | |
|---|---|
| :S | A string property for storing strings of any size, but no null bytes (the default) |
| :I | An integer property for efficiently storing values as integers (1-32 bits) |
| :L | A long integer property for storing values as 64 bit integers. |
| :F | A float property for storing single precision (32 bit) floating point numbers. |
| :D | A double property for storing double precision (64 bit) floating point numbers. |
| :B | A binary property for untyped binary data which may include null bytes. |

# Changing Views

One of the real strengths of MetaKit is that you can easily restructure data that you've already stored in a data file. This is particularly handy given the natural evolution of most programs, where requirements might change over time.

To restructure a view, you simply open an existing data file, and issue a new "`mk::view layout`" command that reflects the new structure you'd like the view to have. If you specify new properties in the original view, they will be added (with a default value, such as the empty string, assigned). If you remove properties, they will be removed from the view in the data file.

# Subviews

Sometimes you will want to attach multiple values of the same property values to a row. For example, in an address book, you might want to record one or more phone numbers for each entry. There are several ways that you might accomplish this. A simple solution is to add properties like "homephone", "workphone", "mobilephone" etc. to the main view. This would give you a fixed number of phone numbers you could store with each entry, and for many entries most of these might be empty.

A second approach is to define a completely separate view, with each row in the new view containing a pointer to the corresponding entry in the address book. For example, if there is a unique "addressid" property in the main address book view, we might define a new view, add records, and search for all phone numbers for a particular address book entry:

```
mk::view layout db.phones "addressid type number"
...
mk::row append db.phones addressid $addressid type home number 123-4567
mk::row append db.phones addressid $addressid type work number 987-6543
...
set rows [mk::select db.phones addressid $addressid]
```

MetaKit provides a third option, using a *subview* of the main view. This allows you to store the phone numbers as a view within the same row you store the rest of the address information. Like normal views, subviews contain zero or more rows, each with the same set of properties, as defined in the `mk::view layout` command. You can refer to rows in a subview by extending the normal row syntax, "`db.viewname!rownum.subview!subviewrownum.`." Here's an example of how subviews can be used:

```
# create the view, including a subview named phones
mk::view layout db.addresses {name city {phones {type number}}
```

```
# add a row (the first one, so it will be stored at index 0)
mk::row append db.addresses name Mark city Ancaster

# store two phone rows in the subview
mk::row append db.addresses!0.phones type home number 123-4567
mk::row append db.addresses!0.phones type work number 987-6543
```

You can use other commands like mk::select or mk::loop that operate on views to operate on subviews as well.

If all you're doing is storing and retrieving data in a subview that is associated with the parent row, subviews may be quicker and easier than using a separate table. But if you plan on doing other searches (e.g. print out all phone numbers), the separate table approach might be better.

## Deployment

MetaKit data files are highly portable, so if you're moving your data file between different platforms (e.g. Mac, Windows, Unix) you don't need to do anything special to make your data file work on the new system.

When actually shipping your application, you can include MetaKit as a normal Tcl extension via a shared library (e.g. mktcl.so or mktcl.dll), or build a static library that you link directly into your executable.

You might also want to look at TclKit (see www.equi4.com/tclkit) which bundles Tcl, Tk and MetaKit into a single platform-specific binary. You can then bundle the rest of your application into a platform-neutral "scripted document". Yours users then need only download your scripted document and the TclKit binary for their platform. This can save you some trouble in distributing your applications on multiple platforms, particularly when you may not have convenient access to each platform.

# Where to Go Next

The official source for MetaKit information, including how to obtain and install the latest versions, is the MetaKit web site, at www.equi4.com/metakit. Make note of the MetaKit mailing list, which can be a source of useful information.

What we've described here are the basic MetaKit commands you'll need in most applications. There are many other commands and options you may find you could take advantage of, and tricks that you may need as your data files grow large. These can be found in the documentation included with the MetaKit distribution (or available via the web site). In particular, you should consult the Tcl reference manual (www.equi4.com/metakit/tcl.html).

## Acknowledgements

Special thanks go of course to Jean-Claude Wippler for his never-ending and tireless efforts in creating, extending, and supporting MetaKit, and for his comments on this document.

# Appendix. Installing MetaKit

If you're familiar with using Tcl and its extensions, then MetaKit should be fairly straightforward to install. But if you're a bit less familiar with Tcl extensions, and perhaps depending on the setup of your machine, you could run into some troubles. Here we'll try to deal with some of the common cases.

## Extensions and Packages

Tcl extensions can be composed of one or more binary shared libraries (e.g. ".so" or ".dll") and Tcl script files. MetaKit is pretty simple in that regard in that it compiles into a single file, a platform-specific shared library (e.g. libmk4tcl.so or libmk4tcl.dll) that contains all of the core MetaKit library including the Tcl API. As you can see by the filename, this extension is known as "Mk4tcl".

Tcl extensions are normally bundled into and distributed as "packages" and loaded into your application via e.g. "package require Mk4tcl". A package is installed in a directory alongside the Tcl distribution, where the directory is given a name reflecting the name and version of the package (e.g. /usr/lib/mk4tcl2.4). Inside this directory there is a special index file (always named "pkgIndex.tcl"), which describes how to load the needed pieces (in this case, just the MetaKit shared library), when an application does a "package require".

## Compiling from Source

If you're building and installing MetaKit from the source distribution, you should have no problems; the extension will be built, and the package properly constructed and installed. This is particularly true if you've installed Tcl on your system yourself.

Problems to watch for here, especially when you're using an existing Tcl version on the system, is that it's a fairly recent version of Tcl, and that the Mk4tcl package is installed in a place where Tcl can find it (you can find out where Tcl looks by examining the Tcl global variable "tcl_pkgPath").

## Pre-Compiled Binaries

Equi4 distributes pre-compiled binaries of MetaKit for a few platforms, so you may also have the option of just downloading the libmk4tcl.so or libmk4tcl.dll file for your platform.

This is just the shared library though, so to use it in your application via "package require" you'll still need to create the package. One option is to manually do this, creating the "mk4tcl2.4" directory where Tcl installs its other packages, and using the Tcl "pkg_mkIndex" command (see the manual entry for this command) to build the pkgIndex.tcl file needed to load the MetaKit shared library.

There is a simpler option though, which you can use because MetaKit is just a single shared library. Rather than loading it into your application with "package require", you can instead load the shared library directly. To do this, first install the shared library in a known location on your machine. In your program, replace the "package require mk4tcl" with the command "load <*path*>/libmk4tcl.so" (or libmk4tcl.dll).