Vlerq + Ratcl = Easy Data Management

Jean-Claude Wippler
Equi4 Software, NL
September 2006

Contents

Abstract

Ratcl and *Vlerq* (rhymes with "flair") are new extensions for Tcl which provide a general data-querying and data-manipulation foundation for in-memory and persistent data, supporting a range of relational and set-wise operations. Data is managed as views, which have the dataflow-like ability to automatically track changes. Views can be stored on file, with full transaction support currently in development. Preliminary memory-use and performance results indicate that the system is very efficient. Some details about the internal design are also provided.

Introduction

One reason why scripting languages are effective is because data is dynamically typed: variables have no type, they can refer to any type of data item. Not having to declare types (or variables) tends to improve productivity and reduce code complexity.

Tcl treats data types specially, in that it maintains the "Everything Is A String model", at least at the scripting level. Due to EIAS, any data can be displayed, printed, saved to file, and sent across a network *as is*.

With this convenience comes a higher overhead, both in memory use and in performance.

A second aspect common to scripting is the presence of a few very general-purpose data structures, called lists and arrays in the case of Tcl. Lists are efficiently indexable, arrays are good at associative lookup.

While lists and arrays are sufficient to write large applications, they are not always optimal. When larger amounts of data are involved or highly structured algorithms are being implemented, it would help to have *stricter* ways to manage that data. This strictness, applied to data types and compound data structures, helps take advantage of a regularity which is present in many application domains anyway.

A distinction can be made between data *items* - for which dynamic type is very convenient - and data *collections* - which consist of items, but which tend to benefit from having a well-defined structure.

By making everything totally dynamic, scripting languages are losing out on various opportunities to improve memory use and performance.

Tcl's EIAS is a trade-off in this respect: there is no way to easily store or transfer data other than as text strings. The lack of type information makes it impossible to take advantage of type even when known to the programmer. Unless one relies on a database library - but that brings its own set of trade-offs in having to leave one language to use another.

This paper presents a conceptual model and an implementation as a pair of extensions for Tcl, which shows how one can get the best of all worlds:

- flexible untyped data handling where needed
- structured data which can easily be passed around
- optional strict data types in (nested) collections
- high performance and low memory use
- easy persistence and support for transactions

The difference with traditional database solutions is that no new language is being introduced - everything is deeply embedded in Tcl, to the point that many optimizations take place without altering the way in which scripts are written.

Data Management

As Frederick Brooks wrote over 30 years ago [1]:

> *Show me your flowchart and conceal your tables,*
> *and I shall continue to be mystified. Show me your*
> *tables, and I won't usually need your flowchart;*
> *it'll be obvious.*

Or: data structures live forever, code changes. This is due to the fact that we tend to store data for a long time, beyond the run time of applications. It's hard to revisit data structure design decisions, once code is written and deployed.

With scripting languages and dynamically typed data this remains just as true as with compiled languages. Changes to data structures still affect a lot of code, especially if access paths and relationships between the data change.

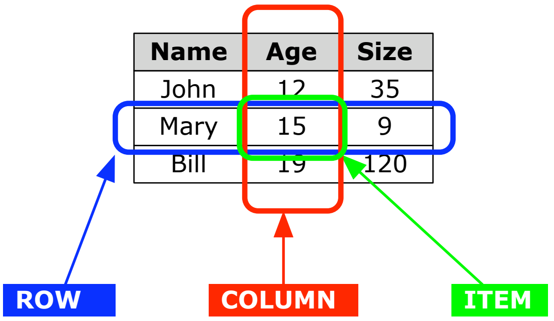
Tcl is very much oriented to element-by-element processing. The *lsearch* command is available for simple queries but all more complex access and manipulation tasks require iteration based on *for* or *foreach*.

Data handling in Tcl is still quite rudimentary.

Ratcl and *Vlerq* extend Tcl with a richer data structure plus a range of *relational*, *set-wise*, and other *collection-oriented* data manipulation operators.

Views

A general-purpose data structure called a *view* is used as foundation for all data manipulation:



Name	Age	Size
John	12	35
Mary	15	9
Bill	19	120

Loosely speaking, a view is like a table:

- a view consists of rows and columns
- rows are accessed by their 0-based position
- columns are accessed by name or by position
- each row / column combination holds an item
- all items in a column have the same data type

The items in a column can be integers, floats, strings, or binary data. With strings, this reverts to being able to store any Tcl value.

Nested sub-views:

But an item in a view can also be a *sub-view*:

---

[1] Frederick P. Brooks, *The Mythical Man-Month*, 1975 - Chapter 9.

<table>
<tr><th>Name</th><th>Phones</th></tr>
<tr><td>John</td><td><table><tr><th>Phone</th><th>Number</th></tr><tr><td>Home</td><td>123-4567</td></tr><tr><td>Work</td><td>345-6789</td></tr></table></td></tr>
</table>

In this example, the main view has one row, with a sub-view in the Phones column. Sub-views themselves might again contain sub-views.

Views are general-purpose containers. There are a few important differences between views and *relations* as used in relational databases:

- rows in a view are ordered and accessible by their ordinal position, tuples in a relation are not
- columns in a view are ordered, whereas the order of attributes in a relation is irrelevant
- views may contain duplicate rows, i.e. a view is a *bag* whereas a relation is a *set*
- views may contain duplicate column names (columns can always be accessed by position)
- relations do not usually support nesting

Views are a superset of relations: a view can represent any relation, the converse is not always possible.

Tcl lists can be represented as 1-column views, and arrays as 2-column views, for keys and values respectively.

Views can be empty, i.e. contain zero rows. They can also have no columns - a zero-column view with N rows is very much like the *count* N.

Meta views

Every view has a *meta view* associated with it which describes the view's field / column structure. For the Name / Age / Size view used as first example, the meta view is:

name	type	subv
Name	S	-
Age	I	-
Size	I	-

Meta views always have three columns, describing the name, type, and sub-view structure of the data view. Each *row* in the meta view corresponds to a *column* in the data view, in the same order.

With nested sub-views, the *subv* column contains a meta view describing the corresponding sub-view.

Views and meta views form an inter-twined combo: even meta views have a meta view - a meta view can be used like any other view.

Ratcl

Ratcl ("Relational Algebra for Tcl") is a Tcl extension which implements views. *Ratcl* is a pure-Tcl extension which wraps the *Vlerq* extension (written in C). In normal use *Vlerq* does all its work behind the scenes: *Ratcl* adds a friendly layer on top to make it all conveniently usable from Tcl.

An example

It's easiest to introduce *Ratcl* with an example (some lines omitted for brevity, output in blue):

```
% package require ratcl
% set v [view {Name Age:I Size:I} vdef {
    John 12 35 Mary 15 9 Bill 19 120
}]
% puts [view $v dump]
  Name  Age  Size
  ====  ===  ====
  John   12    35
  Mary   15     9
  Bill   19   120
```

The main command defined by *Ratcl* is "*view*". It is used to construct and manipulate views:

```
set v [view {Name Age:I Size:I} vdef {
    John 12 35 Mary 15 9 Bill 19 120
}]
```

This takes a list {Name Age:I Size:I} and applies the *vdef* operator to it, giving it a second list as parameter. The result is stored as $v.

```
puts [view $v dump]
```

This uses $v as input, and applies the *dump* operator to it. The result is suitable for display: *dump* pretty-prints its input view. As you can see, the input for *vdef* is used as the list of column names. The ":I" suffix indicates integer values (the default is strings).

Pipelines

The *view* command is also used to create *pipelines* of multiple view operators, where the result of the first operator gets passed on to the second one, and so on. The notation for this is the same as when processing pipes in the Unix shell:

```
view view vop1 ... | vop2 ... | ...
```

So our example could also be written as:

```
puts [view {Name Age:I Size:I} vdef {
    John 12 35 Mary 15 9 Bill 19 120
} | to v | dump]
```

Note that a "to v" operation was added - this assigns the view to variable "v" to mimic the original code.

View access

A common task is to get some data out of views. The workhorse operator for this is *get*, which lets you extract specific items from a view.

- extract one item from row 0, column *Name*:
```
% view $v get 0 Name
John
```
- extract one item from the last row, column *Name*:
```
% view $v get 2 Name
Bill
```
- another way to specify the last row, end-relative:
```
% view $v get -1 Name
Bill
```

- extract one item from row 0, column 0:

```
% view $v get 0 0
John
```

- extract row 0 as a list of values:

```
% view $v get 0 *
John 12 35
```

- extract row 0 as a list of tagged values, i.e. a *dict*:

```
% view $v get 0
Name John Age 12 Size 35
```

- extract column *Name*, as a list of values:

```
% view $v get * Name
John Mary Bill
```

- extract all rows, as a list of lists:

```
% view $v get *
{John 12 35} {Mary 15 9} {Bill 19 120}
```

- extract all items as one list:

```
% view $v get * *
John 12 35 Mary 15 9 Bill 19 120
```

- a shorter way to extract all items as one list:

```
% view $v get
John 12 35 Mary 15 9 Bill 19 120
```

To determine the number of rows, use *size*:

```
% view $v size
3
```

Use *width* to get the number of columns:

```
% view $v width
3
```

The *get* operator can also extract sub-views, meta views, and more.

Both *size* and *width* are in fact also defined as special calls to the *get* operator.

View operators

There are a large number of pre-defined view operators, and new ones can be added in Tcl.

Here are examples of selection:

```
% puts [view $v where {$(Age) > 12} | dump]
  Name  Age  Size
  ====  ===  ====
  Mary   15     9
  Bill   19   120
```

… and sorting:

```
% puts [view $v sort Size | dump]
  Name  Age  Size
  ====  ===  ====
  Mary   15     9
  John   12    35
  Bill   19   120
```

Here is a partial list of operators:

- **Iteration**: collect, index, loop
- **Sets**: except, intersect, union, unique
- **Shape**: repeat, spread, transpose
- **Relational**: join, group, project, sort, where
- **Vectors**: reverse, slice
- **Permutation**: mapcols, remap
- **Synthesis**: concat, pair, rename, tag
- **Display**: dump, html, structure
- **Aggregates**: avg, count, min, max, sum
- **Informative**: meta, names, size, types, width

Grouping and Joining

The *group* and *join* operators are special in that they return nested view structures. The *ungroup* operator is the inverse of *group*.

Grouping operates on a single view - for example:

Name	Phone	Number
John	Home	123-4567
John	Work	345-6789
Mary	Cell	789-7890
Bill	Cell	321-4321
Bill	Home	432-5432
Bill	Work	543-6543

We can group on *Name* and collect the remaining columns into a new *Phones* sub-view column using:

```
view $v group Name Phones
```

The result is this nested view:

<table>
<tr><th>Name</th><th>Phones</th></tr>
<tr><td>John</td><td><table><tr><th>Phone</th><th>Number</th></tr><tr><td>Home</td><td>123-4567</td></tr><tr><td>Work</td><td>345-6789</td></tr></table></td></tr>
<tr><td>Mary</td><td><table><tr><th>Phone</th><th>Number</th></tr><tr><td>Home</td><td>789-7890</td></tr></table></td></tr>
<tr><td>Bill</td><td><table><tr><th>Phone</th><th>Number</th></tr><tr><td>Cell</td><td>321-4321</td></tr><tr><td>Home</td><td>432-5432</td></tr><tr><td>Work</td><td>543-6543</td></tr></table></td></tr>
</table>

Joining can be viewed as “connecting” each row of the input view by looking up the common column(s) in the argument view. It then adds sub-views to hold all the matching rows. Here is an example:

Name	Phone
John	Home
Mary	Cell
Bill	Home
Bill	Work

Phone	When
Home	evening
Fax	weekend
Work	morning
Work	afternoon

We’ll join these and add a *Times* column to the result:

```
set j [view $v join $w Times]
```

The resulting view in $j is:

Name	Phone	Times
John	Home	**When** evening
Mary	Cell	**When**
Bill	Home	**When** evening
Bill	Work	**When** morning afternoon

If there is no matching row in the right-hand view you get an empty sub-view, while matches on multiple rows end up as sub-views with multiple entries. Note that sub-views can deal with all cases in a join without requiring *NULL* support (which SQL needs, and which has some controversial consequences).

The inverse operation of grouping is *ungroup*:

```
view $j ungroup Times
```

The resulting view is:

Name	Phone	When
John	Home	evening
Bill	Home	evening
Bill	Work	morning
Bill	Work	afternoon

So *ungroup* “flattens” a view by tying each row of a sub-view to the remaining fields in the parent view. Rows with empty sub-views are omitted from the result, since there is no data to tie them to.

The combination of the above *join* and *ungroup* is called an inner join in database lingo.

Meta views revisited

The structure of views returned from the different operators is not always identical to the input view(s). This is where meta views play an essential role: each view operator takes care that the meta view of its result is properly defined (with *group* and *join* this is not a trivial task).

So the meta view of any view, whether constructed from data with *vdef* or constructed by any other view operator, is always fully defined. It can be accessed using the *meta* view operator. For example, to obtain a list of the column names of a view, you could use:

```
puts [view $j meta | get * name]
```

This is such a common operation that it has been defined as yet another operator:

```
puts [view $j names]
```

In the above example, the output would be the list {Name Phone Times}.

Iteration

The *loop* operator takes a script which it will then execute for all rows in its input view:

```
view $v loop {
  puts “row $(#) has name $(Name)”
}
```

The *loop* operator defines an array while it is iterating, providing access to all the columns by name for the current row. The name of this array defaults to “”, i.e. the empty array name - hence the somewhat strange-looking “$(Name)”.

Another feature of iteration shown above is access to the current row index via $(#).

There are more ways to iterate, such as *collect*:

```
set map [view $v collect { $(Age) * 2 }]
```

This evaluates an expression and returns a list with one element for each row in the input view.

Custom views

New view operators can be defined in Tcl using the *vopdef* command:

```
vopdef myvop {v args} {
  return [view $v remap $args | concat $v]
}
```

This defines a “*myvop*” operator which takes a list of row positions to prefix to its input view. It could be used as follows:

```
puts [view $v sort | myvop 0 2 4 | dump]
```

This example is not very meaningful, but illustrates how new view operators can be created.More advanced variants of *vopdef* are also available.

Changes

There are a number of operators which take an input view, apply a “change” and return a modified view:

- *set* - sets one or more items in a view
- *insert* - insert one view into another
- *delete* - omit some rows from a view
- *replace* - a more general form of *insert* & *delete*

For example:

```
% puts [view $v set 1 Age 16 | dump]
  Name  Age  Size
  ====  ===  ====
  John   12    35
  Mary   16     9
  Bill   19   120
```

The resulting view is the same as the original, except for one item in row 1, column “Age”.

For *insert* and *replace*, the inserted data must have the same structure as the input view, i.e. it must have the same number and type of columns (but column names need not match, all original names will be retained in the output).

Mutable state

Until now, all views have been treated as values - all operators are purely functional with no side-effects.

This matches the way Tcl lists work: you can only "alter" a list by constructing a new copy.

Just like Tcl, *Vlerq* uses *copy-on-write* and maximizes data sharing where possible by tracking reference counts. But with views and sub-views, this approach is actually taken a bit further: all change operators are implemented on top of *mutable views* - this is a special kind of view based entirely on "smoke and mirrors". Instead of applying changes, a mutable view works by keeping a list of differences along with a reference to the original view. Yet from the outside the effect is indistinguishable from a view which has actually been copied and then modified.

The way to have real *state* changes is just as with lists in Tcl: use variables (scalars or array elements). So you could do something like:

```
set v [view $v set 1 Age 16]
set v [view $v insert 2 $w]
set v [view $v delete 3]
```

Or - equivalently - by using the *to* operator:

```
view $v set 1 Age 16 | to v
view $v insert 2 $w | to v
view $v delete 3 | to v
```

Or even:

```
view $v set 1 Age 16 | insert 2 $w | \
        delete 3 | to v
```

Since multiple changes are optimized in such a way that all changes end up being accumulated into a single mutable view, there is no serious performance penalty for doing this, even repeatedly inside loops.

View references

When state changes are common, it gets tedious to work with views as values and hence to always have to save changes back with an explicit *set* or *to*. This is where *references* come in:

```
% set r [view v ref]
% puts [view $r dump]
  Name  Age  Size
  ====  ===  ====
  John   12    35
  Mary   16     9
  Bill   19   120
% view $r set 0 Size 30
% puts [view $r dump]
  Name  Age  Size
  ====  ===  ====
  John   12    30
  Mary   16     9
  Bill   19   120
```

Explanation: $r is defined as a *reference* to $v. What this means is that $r can be used as a synonym for $v, with one important difference:

> *changes to $r cause $v to be adjusted*

In a way, $r adds a level of "indirection", because it knows about the *name* of variable "v". So accesses will perform a *dereference* and be transparent, and changes will make sure that the new result is stored back into that variable.

References do need to take scoping rules into account - usually global or namespace variables will need to be used to store mutable state.

Dataflow

Views based on references are special in that they will continue to track changes after being created:

```
% set s [view v ref | where {$(Age) > 16}]
% puts [view $s dump]
  Name  Age  Size
  ====  ===  ====
  Bill   19   120
% set v [view $v set 0 Age 17]
% puts [view $s dump]
  Name  Age  Size
  ====  ===  ====
  John   17    30
  Bill   19   120
%
```

Note that $v is the base view, which gets changed and that $s is set to a selection of $v. Changes to the variable "v" affect the view in $s.

Caveat

Knowing that changes propagate, you might be tempted to set a write trace on "s" in this last example, to update a widget for example.

> *That won't work: the trace will never fire!*

It is important to understand why, because this also illustrates how *Ratcl* works under the hood. Let's first examine what $s contains:

```
% set s [view v ref | where {$(Age) > 16}]
where {ref v} {$(Age) > 16}
```

Huh? How can "view $s dump" possibly produce the output shown earlier?

The explanation is that a view is really nothing but a (nested, tree-like) *description* of how to obtain the underlying data. What $s contains is:

- a request to dereference v: "{ref v}"
- rules to apply on the result: "where ..."

So the magic is not in the view as Tcl sees it, but in the operators which are applied to them. A view *as a string* looks trivial - it was the dump operator which caused the *Vlerq* extension to interpret that string in such a way that the information got extracted.

Back to the explanation of why traces won't fire:

> *$s does not change when variable "v" changes*

In other words: the *description* remains constant - it is the data behind that description which changes. In a way, a view in Tcl is a little program, one which the "view" command and all its operators happen to know how to execute - so it is not the program which changes, but its output when executed.

Real dataflow!

So how can we have real dataflow in Tcl, i.e. changes to views "rippling through" to all views defined on top of them, and then possibly firing custom scripts to act upon such changes?

The answer lies in distinguishing a view *definition* from its *data* - the "little program" mentioned above.

To make dataflow work, *Ratcl* defines an extended form of the *to* command:

```
view ... | to <name> operator...
```

An example:

```
% view v ref | to t get * Name
% puts $t
John Mary Bill
% set v [view $v set 0 Name Joe]
% puts $t
Joe Mary Bill
```

Now, $t no longer contains a view description but all the rows of column "Name". So $t is a list of actual values. With actual values, the problem of description versus data is gone.

This also works for the size of a view:

```
% view v ref | to t size
% puts $t
3
% set v [view $v delete 1]
% puts $t
2
```

And for meta-view details like column names:

```
% view v ref | to t names
% puts $t
Name Age Size
% set v [view $v project Age Name]
% puts $t
Age Name
```

Now dataflow will work fine - in the sense of causing traceable state changes in Tcl - when a view change leads to actual data getting stored in a Tcl variable.

Behind the scenes a lot of caching and dependency tracking goes on to maintain the perception of automatically updated dynamic views.

Sub-views

Dataflow also works with sub-view changes: when a sub-view is changed, it is the parent view which gets modified. Assuming $w is a reference to a suitably nested view, the following changes are equivalent:

```
% view $w get 0 Phones | set 1 Phone 234-5678
```

and

```
% view $w set {0 Phones 1} Phone 234-5678
```

In other words, the *set* operator accepts a list as row position, which is then interpreted as access *path* into a nested view. It does not matter whether the path has been traversed explicitly using *get* or is being specified as a *set* path. Both cases end up doing the same: record a sub-view change in its parent view.

Barriers

There is a limit to dataflow, however:

```
% view $w sort | set 0 Name Alan
```

The above will *not* change $v - changes do not travel upwards in a calculated view hierarchy.

While there are a variety of scenarios where such a change can be meaningful, this is not always the case. For this reason, the current implementation of *Ratcl* considers almost all view operators to be a dataflow *barrier* - only changes to the top-most view and all its sub-views will "percolate" back through to view references. In practice, this means that changes only traverse back through immediate references and through sub-view accesses, i.e. the *get* operator.

In cases such as *sort*, changes act by inserting a new mutable view after the sort, and applying the change there. You *can* change a sorted view, since it acts as a value, but the change will not have side effects: it will only be visible in the view returned by the change.

Changes to derived views do not break the value-based semantics of views. All changes are implemented in a virtual way: a change to a sorted view does not lead to data copying, it gets recorded as a difference relative to the (virtually) sorted view.

Persistence

Views can be saved to file and loaded back later on. Loading is instant, because *memory-mapped files* are used to bring data in only when actually used. Views are not "read" from file in the traditional sense but swapped in using hardware-assisted on-demand paging. In modern operating systems, this is very efficient regardless of file size - even when the data file size exceeds the amount of available RAM.

Here is how to save a view to file:

```
view $v save myfile.db
```

And here's an example of how to use it again:

```
puts [view myfile.db open | dump]
```

To serialize to a Tcl channel instead of a file:

```
view $v save -channel $socket
```

To serialize a view as a binary string:

```
set data [view $v save]
```

To read a view from a channel (reads to EOF):

```
puts [view $socket channel | dump]
```

To use a saved string as a view:

```
puts [view $data load | dump]
```

Since views can contain nested sub-views, the above also works for collections of views. One way to save multiple views is to create a view with one row and several different sub-views, and to then save/load that top-level view instead.

Data files created in this way are very compact, portable across architectures with different byte orders, and in fact fully compatible with the Metakit embedded database library [2].

There is a *freeze* view operator, which takes an arbitrary view, converts it to an optimally-packed binary string, and then uses that string to reconstruct an identical view. It is defined as follows:

```
vopdef freeze {v} {
  return [view $v save | load]
}
```

This is useful to create a permanent copy of a view which was constructed from other views with other view operators, and it also allows you to release the memory used by the original view and replace it with a maximally compact in-core representation.

Persistence includes all sub-views: nested views can be saved in the same way as simple "flat" views.

Transactions

With only a snapshot-like load/save mechanism, *Ratcl* could not be called a "real" database - one of the distinctive features of a database is that it efficiently deals with large amounts of data, and loading *all* data into memory before use will not scale well.

However, due to the use of memory-mapped files, "opening" a data file is actually already instant in *Ratcl* regardless of the amount of data in that file. What remains is the requirement to save changes quickly and 100% reliably under all circumstances.

The database term for this is *ACID-complian*ce - ACID stands for Atomicity, Consistency, Isolation, and Durability [3].

It should now become clear why the decision to store view changes as differences was made: to commit a transaction, what we need to do is save those differences in such a way that they get re-applied on open.

And this is precisely how a recent implementation of *Ratcl* works: all changes are collected as differences in memory, and on commit a single write operation is used to append the total set of changes in one go.

The file format used is the same as in Metakit - it was specifically designed to support atomic appends. This has another advantage on modern journaled file systems that one does not need to use *fsync* system calls to enforce fail-safe storage. The way writes are performed is such a way that on a properly configured journaled file system such as Ext3, changes to a data file are guaranteed to either complete fully or to be rolled back fully after catastrophic system failures (including power outages).

The file format has also been designed in such a way that *Ratcl* can deal with *multiple-readers / single-writer* without any file locks. The end of the file determines the state of the data file, and since that only changes in atomic ways, readers always see a consistent state.

For less robust file systems, *fsync* system calls can be inserted to enforce absolute durability. There is a performance trade-off: with *fsync*, the rate of commits is bounded by the rotational latency of hard disks.

Due to the way in which all changes get appended to a file, files will grow after each commit. There is a need to compact files so that data areas which are no longer in use can be reclaimed. This is currently not implemented - the simplest way to compact a file which has grown for some time is to save it to a new file and replace the original (this needs to be done while the data file is not is use by others).

More sophisticated solutions are possible to reclaim unused data, but none of these have been implemented or even explored so far.

Note that files grow in size proportional to the amount of new data committed, not to the size of views or of columns in those views, as in Metakit (which *does* reclaim unused free space, by the way).

The status of transactions is still very much in flux, now that dataflow capabilities are being added to *Ratcl*. For this reason, the most recent version of the code does not support transactions any more. See the *Current Status* section later on for more details.

Efficiency

There are a different sides to efficiency.

Memory use: data

In Tcl, memory use for various data types and structures is as follows (in a 32-bit machine):

- integers and floats: 24 bytes per value
- strings: 24 + length of string in UTF-8
- lists: 24 + 4 * #-of-items + items themselves

More memory is consumed when both the internal form of a value and its string version are in use.

With *Ratcl*, memory use is as follows:

- integers: 0..64 *bits* per value
- floats: 4 bytes, doubles: 8 bytes
- strings: 4..8 + length of string in UTF-8
- views: fixed overhead of around 32 bytes per column + the items themselves

---

[2] See the Metakit home page at http://www.equi4.com/metakit.html

[3] See the definition of ACID in Wikipedia at http://en.wikipedia.org/wiki/ACID

As you can see, the overhead per item is low for most data types, especially for numbers.

Memory use: operators

In *Ratcl*, most view operators are virtual. They do not copy their input data but track the original as well as any additional information they need - the actual requirements depend on the operator.

Some examples (overhead in bytes):

- **sort** - 4 x # of rows in the original view
- **where** - 4 x # of rows in the result view
- **join** - roughly 8 x total # rows in both views
- **group** - roughly 8 x # rows in original view

There is also some fixed overhead for each view, but it is not related to the number of rows in the view.

Constructing new views from existing ones is very efficient in memory use: views with a million rows can often easily be dealt with in today's standard desktop machines.

Speed: item access

The time to access an item in a view depends on a number of factors:

- data type: it takes more time to fetch a (variable-sized) string item than a number
- view nesting: because many views are virtual, access into a deeply nested view will be slower than access to a base view with data

Timing results show that for a base view, access to an item is about 3x as slow as *lindex* on a list in Tcl. This is a bit tricky to estimate, since Tcl bytecode-compiles *lindex* whereas the *view* command always has to go through a C extension call.

Compared to Metakit, i.e. the Mk4tcl extension, *Ratcl* access is at least 3x faster. No timing comparisons with other database systems have been made so far.

Speed: operators

Performance-wise, view operators tend to be as fast, or sometimes considerably faster than Tcl. One reason for this is that view operators are coded in C and do not need to go through the Tcl C interface for each item access.

Sorting is between 2x and 4x faster in *Ratcl* than it is in Tcl (depending on the data types) for simple keys. For compound keys, *Ratcl*'s sort is faster still. Compared to Mk4tcl, sorting appears to be around 15x faster with *Ratcl*.

Note that Tcl's sort is coded in C, it too does not need to go through the C interface for each item access, although it does internally call the C equivalent of *lindex*.

Grouping and joins have no direct equivalents in Tcl - other than coding them as scripts, which would be at least an order of magnitude slower.

Nevertheless, it is possible to get an impression of the very high performance of both *group* and *join* in *Ratcl* by noting that it takes roughly 1 second to apply these operators to views with a million rows (in the single-key case). These timing results were obtained on a 3-year old machine with a 1 GHz PowerPC G4 CPU - more modern hardware will surely exceed this.

Why is Ratcl fast?

It might be surprising to see such good results for a relatively simple data management extension in Tcl, but there are in fact two explanations for this and they appear to complement each other:

First of all, both *Ratcl* and *Vlerq* were designed from the ground up for high performance. All the key loops are coded in C, so there is no Tcl overhead at all: a view operation is often a single call to a C primitive inside *Vlerq*.

The second reason is the choice of internal data structures: *Ratcl* and *Vlerq* are based on a highly *vectorized* architecture. Not only are all items in a view represented in an efficient form, they are also organized in *column-wise* form, i.e. the opposite of how one normally tends to look at collections of structured data. So while views look and act like rectangular structures, they are in fact internally stored as (a few) columns with all items end-to-end, i.e. in vector format.

This has a huge impact on how various algorithms can operate - many of them have been optimized to take maximum advantage of the *locality of reference* which is inherent in a vectorized data representation. As it turns out, modern CPU's really soar when this is done properly: one of the limiting performance factors nowadays is memory access. The trick is to get lots of relevant data into the CPU's hardware caches. Then, with vectors, iterating across all their elements in sequential order leads to massive performance gains.

Surprisingly, there seem to be only very few other database packages which use this same "inverted" column-wise approach: MonetDB [4] and Kdb [5]. Both offer blazing performance, by the way.

Internals: Vlerq

Vlerq is the engine underneath *Ratcl* which implements the concept of views and many basic operations on them. It is written in C and a large part of it independent of the Tcl language and interpreter.

---

[4] See the MonetDB home page on the web, http://monetdb.cwi.nl/Home/

[5] See the Kx Systems home page on the web, http://kx.com/

Nevertheless, *Vlerq* was designed to work optimally with Tcl - and several design choices reflect this.

Value-based data handling

Tcl is a *value-based* language: scalars and containers are not modified but copied and then modified, so that changes to data do not have side-effects. This is done using *copy-on-write* with the optimization that data structures which are not shared can omit the copy and be modified in-place. All side-effects in Tcl come from changes to variables (and procedures).

Views adopt this same approach: a view is a value which will never "suddenly" change. All modifying operators work by (conceptually) creating a copy and then applying the changes to that copy. To model a permanent state change, you have to assign the new value to a variable.

Since views can be nested, the value-based approach is carried through all the way: a change to a sub-view will not change the sub-view itself, *nor the parent view* owning this sub-view. Instead, a new parent copy is made, with the sub-view change applied to it.

One reason why this approach is not very costly in terms of speed or memory use, is that all changes are tracked as differences. Change overhead is relative to the amount of change, not the size of views.

A second reason why this works well, is that all changes can share the same original view - regardless of its size or the level of sub-view nesting. There is even more sharing between views than with Tcl lists.

Because original structures are kept intact, this also means that views loaded from file can be modified without immediately modifying the file. Better still, views do not have to be loaded to be changed: all we need to do is remember the differences!

Column-wise data representation

Vlerq uses an unusual approach to representing tabular data: a table is represented as a collection of *columns*, not rows. The "N-th row" is a virtual concept, it gets mapped internally to the N-th item in each of the columns. *Vlerq* fully hides this mechanism and presents a normal rectangular row/column model.

The column-wise approach has been used in Metakit for many years. When properly used, it can lead to very efficient operation. Having data in "inverted" column-wise form is a little as if every column is an index - *all* the time, and without requiring the construction of separate indexes.

When rows are kept in sorted order (sorted on one or more key columns, that is) then binary search can be used to locate rows by key value.

A "blocked" view structure can be added on top of these purely linear views to get the same locality-of-reference benefits as B-trees. This too is shamelessly copied from Metakit. It is very effective is keeping performance high when views contain many rows.

For those cases where brute force scanning is not good enough, lookup maps can be created which map a key value to a row number (using a second view with two or more columns). This is the column-wise equivalent of a secondary index. *Ratcl* supports this approach, but like Metakit it is not yet automatic.

Virtual operators, no data copying

The term "view" was chosen very deliberately in Metakit and in *Vlerq*: views can be either real data (i.e. a set of columns) or virtual. In the latter case, the view access function will return values *as if* the data is actually stored that way, but it can be represented in a completely different way.

Or not at all, even. For example, the *sort* operator takes an input view and returns a new view. That new view is not a copy of the original in a different order. Instead, a sorted view consists of a single column of integers, used as permutation over the original data. So when you get the N'th item in a view, the sorted view will translate the "N" to another index via the permutation vector, and then ask the original input view to return that value instead.

This has profound implications for memory used and for data access patterns in general.

When you load a view from file, only a small amount of administrative information is actually set up - the returned "view" is really more an algorithm which knows where to read the data from *when needed*. To sort such a view, a scan through all keys is used to determine the sort order. This requires a scan over a few columns of the view on disk (a merge sort, actually), and an in-memory integer array to store the sort permutation. No keys are copied, nor is any other data even read off the disk.

Only when the resulting sorted view is accessed, will the actual data transfer off the disk take place. Again, since the file is memory-mapped, this happens with no copying until the very last moment: when the value is needed in the Tcl code.

This explains why it takes so little overhead to "load" a view from disk (not much happens on open). It also explains why iteration over all rows in a view has the same overhead regardless of the number of columns in that view (only the data needed is read).

Views are highly virtual - they often simply represent the *knowledge* of how to access data. Actual accesses and I/O only takes place when the data is *used*. With many set-wise operations, those accesses then often take place in large streaming batches, i.e. taking maximal advantage of the vector-like storage form.

Dual representations & delayed evaluation

With Tcl, there is one additional level of indirection. As shown before, views are described (via *Ratcl*) as trees of simple accesses and view operations.

These descriptions are in string form (or as lists) and do not even trigger the underlying access mechanisms of *Vlerq* right away.

You could open a file, load a view, and sort it using:

```
  set v [view dat.db open | get 0 addr | sort]
```

Note that at this stage not a single data access operation has been performed, $v will contain the string:

```
 sort {get {open dat.db} 0 addr}
```

A trivial string, created instantly.

> *Note: This is a slight misrepresentation of how things really work (the open and get operators are performed right away), but for the sake of argument, let's assume that the above is accurate.*

Now, we can do:

```
 puts [view $v dump]
```

At this point, the situation changes completely. We need the actual size, structure, and contents of the view to be able to dump it. All the above will cause *Vlerq* to convert these strings to an internal view representation. First the file will be opened, then all the keys in the "addr" sub-view will be traversed to sort them, then all data will be accessed, and lastly all values will be formatted and dumped as a string.

Dataflow implementation

With this approach, dataflow turns out to be simple: when changes in underlying variables are detected, *Vlerq* will go through all the Tcl_Obj's of views depending on that variable, and will invalidate their internal representations. That's enough, because subsequent accesses will reconstruct views again as before, but based on the variable's new contents.

There is a fair amount of complexity in tracking all the dependencies between views and their operations, but this does describe the basic mechanism.

Current Status

Over a ~~million~~ dozen more-or-less complete rewrites have been performed for *Ratcl* and *Vlerq* in the past *several* years as the software evolved substantially.

About five iterations of *Vlerq* have been developed:

- version 0 - a Forth-like VM in C++ (vintage 2000)
- version 1 - VM coded in C, added Thrill language
- version 2 - new "boxes & cells" data structures
- version 3 - new Tcl-specific version, started in 2006
- version 4 - simplified fast C core, separate Tcl layer

Most of the details described in this paper were implemented in version 3, but a rewrite was required to implement a more general dataflow mechanism.

The code in version 3 is quite stable, it passes some 500 tests coded in Tcl. The code can be downloaded as a source snapshot - it includes *Ratcl* and *Vlerq* and is packaged as a TEA3-compliant Tcl extension.

The performance figures mentioned so far are based on the v3 implementation.

Version 3 was also (somewhat frivolously) called "Take 42", because that's about the number of attempts it took to write this implementation. Details for downloading can be found on the web at:

http://www.vlerq.org/vqr/347

There is no documentation for version 3, the best source of information is probably the set of tests in the tests/ subdirectory.

Version 4 is in "frantic" development mode right now. The core data structures have been extended to handle dataflow from the ground up. Some 60 tests are working, more are being added every day.

The major task ahead is to carry all the functionality of versions 2 and 3 over into version 4. This is dependent on first getting dataflow working 100%.

An early version of difference-based transactions was implemented in version 2, it will need to be adjusted to work again in the current v4 code base.

The latest news about *Ratcl* and *Vlerq* can always be found on the website, at http://www.vlerq.org/

Acknowledgments

First and foremost, I would like to thank Mike Doyle and Eolas, Inc. for funding my work on *Vlerq* since early 2005. Without their generous support, and the permission to release all work as open source, I could never have invested as much time in this project.

I am also grateful to Brian Theado for his continued feedback and encouragement. It takes some serious perseverance to track a highly-evolving project like this, as it moved from being a minimally-functional research project to a reasonably usable first development version.

Lastly I would like to thank Mark Roseman for his patience and help in going through this paper and offering many valuable suggestions.