

Relational algebra for Tcl: introducing Ratcl and Rasql

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

ABSTRACT

There are a number of ways to manage data in Tcl, from native lists and arrays to various database bindings. The choice involves trade-offs regarding persistence, robustness, performance, memory use, query capabilities, scalability, portability, standards conformance, and convenience. This is an issue even for simple scenarios, given Tcl's limited support for data structures. A new approach will be presented which is based on relational algebra, supplied through two packages: Ratcl, which provides a relational algebra extension that fits naturally with Tcl. And Rasql, which provides a layered SQL interface for those who prefer it. Both are based on over a decade of experience gained with Metakit's vector-oriented internal data model, and use a new very compact and efficient C-coded engine called Thrive. Ratcl introduces a terminology and set of conventions which minimize the impedance mismatch caused by having databases added on instead of native persistence and query support, while Rasql takes this one step further to map standard SQL queries onto Ratcl. Working examples will be presented, along with performance results so far. This covers the first phase, which focuses on access and querying. The second phase is work in progress and will be briefly described - it deals with modifications, transactions, and multi-user scenarios.

Introduction

Tcl has a range of mechanisms to deal with data, both in-memory and on-disk. One of the more unusual and very powerful aspects of Tcl is that (almost) “everything is a string” (EIAS) as far as the programmer is concerned.

This is both a blessing and a curse. The total lack of inherent type with EIAS that makes it very easy to quickly write code, can also be a cause for trouble:

- Type-less data can lead to code where bugs are caught later in the development process.
- Fewer opportunities to work with highly optimized data representation formats.
- No simple solution for missing values, i.e. null as being distinct from the empty string.

The lack of explicit type translates directly to the lack of explicit structure, i.e. compound types (records). While Tcl offers some very convenient mapping such as the new “dict” convention in 8.5, this type is not as easily enforced or stored truly efficiently on file.

When large volumes of data are involved (more than can conveniently be held in memory) or when high performance data manipulation is required, the EIAS approach by itself tends to lead to a lot of Tcl coding to deal with the unwanted consequences. This is usually just about the time when people start looking at databases as a way to address these issues.

There is something odd going on: as a language, Tcl offers amazing productivity gains when it comes to developing large-scale production software, but when a substantial amount of data is involved, much of the benefits are left behind as the coding switches to a

very different database style, such as SQL.

This paper presents a conceptual model based on Relational Algebra (RA) and shows how it can be embedded in Tcl in such a way that the benefits of scripting and quick ad-hoc coding remain, while the data gets managed in a completely new way, with high performance and persistence thrown in for free.

The Ratcl and Rasql extensions described here are part of a larger research project called “Vlerq”, which will also be presented briefly later on.

So what’s the problem, really?

The main reason why data storage has so many implications for programming is *copying*.

Programs are strange beasts: when launched, they start with a completely empty slate – all data processed by a program needs to be brought in, either directly from file or via a database layer. Worse, all new data, and all results produced by the program need to be saved back to “persist”. Do nothing, or crash, and the data will vanish. Imagine us working that way, knowing nothing when we wake up, and forgetting everything when we go to sleep!

So what most programs do, and have been doing for decades, is to create mechanism to facilitate this task of “fetching” and “storing” data, or “loading” and “committing” in database parlance.

The mindset that goes along with this is very deeply entrenched in most programming lan-

guages. One possible exception is Smalltalk / Squeak, where the system itself loads all data on startup and saves it again on quit – treating code and data uniformly.

For a database to be usable in Tcl we expect it to be good at fetching the data we need, and good at saving it back robustly and efficiently when changed.

Let's examine the different existing approaches to data storage, before attempting to offer an alternative.

Flat files

The simplest form of data storage by far is to dump everything to file, and restore it all in full later. The term “flat” is used, because in most programming languages this tends to destroy all structure, i.e. inter-data relationships.

In Tcl, structure can be saved on file for free. If you store a list, it'll come back as a list. This is one of the immense benefits of the EIAS approach of Tcl. So in a way, Tcl is actually much better equipped to work with (flat) files than most other languages.

There are drawbacks to flat files, though. For one, you have to dump and restore all data at once. There is no easy way to work with subsets (especially in terms of saving only the changes back). This makes the dump/restore slow as more data is involved, and means a copy of all data has to be held in memory.

Another problem is robustness. When saving changes, you have to be very careful not to lose all data altogether if the system were to crash or be switched off at just the wrong moment in time.

Yet another problem is evolution. How do you deal with old files when a new version of the application requires the data format to be extended in some way?

All of a sudden, flat files turn out to be not so trivial anymore. We're getting bitten by the fact that having data on-file and in-memory as totally disjoint forms of the same is not that convenient after all.

Relational databases

The next solution is to adopt a database of some kind. By now, this is almost always a relational database, based on decades of work leading to a very sound theoretical foundation for both the way to structure data and the way to manipulate it.

There are many relational database implementations, of varying complexity and sophistication. The most common ones in Tcl are probably Oracle, PostgreSQL, MySQL, and SQLite. The latter is quite interesting because it is embeddable, i.e. part of the application, whereas the others mentioned here are client/server solutions using a separate process or even machine.

With a relational database, all the problems mentioned for flat files are solved. Access and modification in subsets of the data are easy and quick. Data no longer needs to be loaded on startup or saved on exit. Changes are saved as transactions, so that fail-

ure is completely controlled: either a set of changes makes it into the database or it does not – there is no intermediate or inconsistent state.

The robustness of relational databases is summarized with the “ACID” acronym: changes are Atomic, Consistent, Isolated, and Durable. It's good to be able to truly rely on a database – after all, a crashed program can usually be restarted, but damaged and inaccessible data is essentially unrecoverable.

This comes at a price, however.

Data in a database can be orders of magnitude slower to manipulate than in-memory data. Try comparing a relational “join” with a Tcl array lookup, which are more or less the same operation, in abstract terms.

Apart from speed, databases tend to highlight the huge difference between in-memory data and on-file data. Something as simple as “\${a}\${b}” in Tcl ceases to be available. Instead, you're faced with, say:

```
[$db {select * from a where key = '${b}'}]
```

... with not just a performance loss but also the issue of accurate quoting when \$b is an arbitrary string.

In theory, relational databases are wonderful. In practice, they can be a pretty lousy fit for programming languages.

Other ways to store & manage data

There are a number of other solutions to dealing with large amount of persistent data.

OODB – To overcome the impedance mismatch between databases and programming languages, a number of object-oriented solutions have been built. The idea is to treat everything as an object, and to then add a mechanism whereby objects transparently move between their in-memory form and a “backing store”, using techniques such as “pointer swizzling”.

A hybrid is “object-relational” mapping (OR), where objects are mapped to records in relational databases.

The OODB approach will not be explored further – one reason being that Tcl uses EIAS as basic model, not OO. But more importantly, OODB suffers from a major flaw when compared to relational databases: they tie the navigational access model to the data structure. In other words: when using an OODB, you have to make choices on how the data will be accessed, whereas the relational model separates the data structure from the way it is used. This is a very fundamental issue, at the heart of many OODB vs. RDB debates.

XML – Another approach is to fully abandon the relational model and treat everything as a hierar-

chy. XML was designed as general-purpose interchange format, and is now occasionally touted as solution as the model to use for storage and manipulation of that data as well.

There is little benefit to doing so, actually. Apart from the fact that it does not address the main issue of avoiding the gap between on-disk and in-memory formats, the main drawback is that by ignoring inherent repetitive structure in large data-sets, it prevents a number of optimizations and notational conveniences from being used.

Lastly, XML data can in fact very efficiently be represented via the relational model.

Berkeley DB – This represents a range of different database implementations actually (such as gdbm). The model used is the key/value association. With the *DBM packages, all data is stored by key, and looked up by key, plus the ability to traverse all keys.

This can be summarized as the persistent equivalent of Tcl arrays.

The speed of keyed access can be quite high, due to the use of hashing, although that tends to break down when large numbers of accesses are performed, where hashing leads to excessive disk seeking.

This approach is not used much, despite the fact that it has been around for ages. One reason is no doubt that richer data structures are often needed, and that as with OODB and XML solutions it often is useful to be able to navigate through the data in other ways than by key. A request such as “find all keys X for which the value is 123” ends up traversing all data.

Metakit – Metakit is a mix between the flat-file, relational, and hierarchical database approaches. It uses an inverted column-based format for efficient brute-force searching across all data, and uses the “stable storage” algorithm for transacted changes.

The Metakit database is a bit of everything and a bit of nothing. It has been used as basis for a relational SQL layer, although the Tcl binding does not really expose all functionality of the core.

The basic goal was to try and combine the powerful relational database concepts while using a column-wise internal structure for performance. To put it another way: Metakit presents a row-wise interface to what is essentially an “inverted” format. It favors fast access/searching, at the cost of slower updating.

Searches in Metakit can use hashing or binary search, but they are usually done by brute force. The reason this works so well is that copying is avoided to an extreme degree. Iterating over one field in all rows often outperforms other databases, even when they use indexes (up to a point, of course).

Brute force searching also works well with imprecise searches, i.e. “globs” and regular expressions, where a full scan is usually needed anyway. In Metakit, text searches are cheap.

One consequence of the inverted design is that data

structures can instantly be extended or modified. Adding a field to all rows is a matter of adding a single column internally. This encourages gradual development – extend the data as your code grows, instead of designing it all up front. This is a great match for the dynamics of scripting.

But Metakit is not perfect. Its Tcl binding does not expose some of the more advanced capabilities of the underlying engine. And although quite snappy, the design is far from optimal in terms of performance.

Lastly, Metakit’s documentation is lacking. It takes some work to get the best mileage out of the system.

Home grown – There will always be data storage solutions that are custom-designed for a specific task. The challenge of any new solution is of course to try and offer sufficient performance and flexibility to cover an increasing number of these cases. The trend towards using “standard” solutions appears to be increasing, no doubt because home grown code is much more work to maintain, and because more and more open source alternatives present themselves.

Client/server – Lastly, one could say that the easiest way to use a database is to not use one at all. Instead of incorporating code for storing and manipulating data inside the application, the alternative is to simply connect to a database on a remote server. This relies on permanent network connectivity – an obvious trend, as the rise of websites with databases behind them shows.

Looking for alternatives

Wouldn’t it be great if we could *somehow* combine SQL’s relational foundation with Metakit’s column-wise performance and embed it all really cleanly in Tcl?

This is precisely the aim of Ratcl and Rasql.

The strength of SQL is that it has a strong relational foundation that is extremely effective (even though some will argue that SQL is severely flawed). It is a great benefit to be able to specify data processing tasks in a non-procedural way, i.e. in terms of what needs to be done, not how it is done.

Not only is it easier to say “find all the names of the part numbers I have on this list” than “go through each item on this list and lookup the name associated with in the parts catalog”, it also leaves more room for the underlying code to choose between different implementations. In cases where performance is not at a premium, the benefit of not having to spell out the details surely does simplify programming.

Then again, the SQL world is rife with examples where changing the order of a request makes a huge difference in performance, or where one is

expected to add an index briefly for use in a specific task, and drop that index again to avoid hampering other tasks. The last thing we need is a system where we have to fight and apply counter-intuitive tricks to get good performance.

If you think this is a minor issue, think again: people abandon SQL all the time due to the unacceptable performance they get (for whatever reasons).

Metakit proves that an inverted column structure has the ability to outperform traditional databases, sometimes by an order of magnitude. Examples are known for each and every database mentioned so far, where Metakit was able to perform the same task an order of magnitude faster. The very high-end “Kx” commercial database using a similar design shows that the limits of scalability and performance have not yet been reached, not by a long shot.

The challenge ahead, is to embed these techniques into Tcl in such a way that one stops thinking in terms of getting data “out” of a database and storing changes back “in”. Better still, we should try to create a system whereby the whole concept of a “database” separate from the language fades away.

This is similar to the way Tk has pushed “graphics contexts”, “ports”, “screen coordinates”, “refresh”, and “updates” out of the mind of the application programmer. We don’t think of Tk as a place to copy Tcl data to. We create a view hierarchy in terms of widgets, and then events do the rest.

There is a tremendous opportunity here. A lot of effort in programming deals with moving data around, altering its shape and structure a bit, and transforming it – often in very simple ways. At every point, we have to think where to copy data from, what variables to put it in, and how to deal with the end results – on-screen and on-disk.

Already, Tcl has many types of data collections. Internal data, such as channels, widgets, commands, as well as external data, such as returned from glob, stat, events, I/O.

Already, we lack a consistent way of combining this data. An example of this is: give me a list of a read-only files in a directory. In Tcl, we have to get a list (glob), iterate over them (foreach), check the file’s attributes (file stat), and generate a list with results (lappend). Why can’t we join the glob to the file stat and apply a condition?

Relational algebra provides a simple formalism, which is every bit as powerful as SQL (more so, some will say), and which lets us specify (as opposed to spell out) what needs to be done.

To get there, we need to “let go of the data”, i.e. stop thinking in terms of storing it in variables. Instead, we need to set up our processing in terms of operators (and use variables to manage those structures).

We need to let Tcl do what it does so well: glue.

Introducing Ratcl

The Ratcl extension for Tcl takes a first step towards a non-procedural approach to programming.

To use Ratcl, you have to be prepared to place all data under its control. Doing so will give you low memory consumption, persistence, and performance in return. Data in Ratcl can be manipulated through relational operators (join, groupby, and so on), set operators, expressions to produce calculated results, conditions to define subsets, and sorting.

The central concept in Ratcl is the “view” – think of it as the widget of the data world. A view is a tabular structure with the following properties:

- Views consist of rows, indexed by position.
- Views consist of columns that can be referred to either by name or by position.
- At every (row,column) position is a data item, which is either a basic value such as an integer or string, or a nested “sub-view”.
- All items in a column are of the same type.

The above terminology will be used in the rest of this paper, but usually very similar designs underlie most database systems. Here is a comparison with some familiar concepts:

- SQL’s “tables” are similar to views – they do not support positional access, usually, nor nested sub-views. In SQL, rows are called records and columns are called attributes. Views are indexable, they can also represent result “rowsets”, there is no need for cursors.
- The “relations” of pure relational database theory differ from views in that neither positional access nor order is supported, for rows as well as columns.
- Tcl arrays (and Python dictionaries) are very similar to a view with a “key” and a “value” column. However, views treat keys and values on equal terms, and allow either of them to consist of multiple columns.

It might be tempting to see views as matrices of rows and columns, but this is in fact not such a good idea. For one, matrices are uniformly typed, whereas each column in a view can hold different types of data. The other reason is that views will be extended later to support dimensions independent of row structure (so you could have a 3-dimensional space of rows of arbitrary complexity, not just single values).

Views are the central interface between Ratcl and Tcl. In Tcl, a view is a command object. You create a view explicitly and fill it with data in one command:

```
% set V [view A B C \  
          { a1 b1 c1 a2 b2 c2 }]
```

To dump the view in Tcl, simply execute the command with no arguments:

```
% $V
  A   B   C
  --- --- ---
  a1  b1  c1
  a2  b2  c2
%
```

As you can see, V was a view with two rows and three columns, named A, B, and C.

Yes, V *was* a view, not *is*, as you can see here:

```
% $V
invalid command name "::vlerq::o::1"
%
```

Views are command objects in Tcl, but they require a slightly modified style to be usable transparently in Tcl. The details of this will be explained later, for now it is sufficient to note that with view objects, you should use “vset” instead of “set” when storing their name in a Tcl variable (or array element). To repeat:

With views, use “vset” instead of “set” !

This idiosyncrasy is only needed in Tcl, btw. Other languages can handle views like any other object.

With these preliminaries out of the way, let’s see what Ratcl has to offer.

A little tour

Ratcl includes a wide range of view operators. A few basic examples are given here. See the Ratcl pages on the web for more complete examples and some preliminary reference documentation.

Let’s assume the following views have been defined:

```
% $R
  A   B   C
  --- --- -
  a   b   c
  d   a   f
  c   b   d
% $S
  D   E   F
  --- --- -
  b   g   a
  d   a   f
% $T
  A   B   C   D
  --- --- --- -
  a   b   c   d
  a   b   e   f
  b   c   e   f
  e   d   c   d
  e   d   e   f
  a   b   d   e
% $U
  C   D   E
  --- --- -
  c   d   e
  c   d   f
  d   e   f
%
```

Then we can do things like:

```
% [$R product $S]
  A   B   C   D   E   F
  --- --- --- --- ---
  a   b   c   b   g   a
  a   b   c   d   a   f
  d   a   f   b   g   a
  d   a   f   d   a   f
  c   b   d   b   g   a
  c   b   d   d   a   f
% [$T project {A B}]
  A   B
  --- -
  a   b
  b   c
  e   d
% [$T if "B > 'b'"]
  A   B   C   D
  --- --- --- -
  b   c   e   f
  e   d   c   d
  e   d   e   f
% [$T join1 $U]
  A   B   C   D   E
  --- --- --- --- -
  a   b   c   d   e
  a   b   c   d   f
  e   d   c   d   e
  e   d   c   d   f
  a   b   d   e   f
% [$T join0 $U]
  A   B   C   D
  --- --- --- -
  a   b   e   f
  b   c   e   f
  e   d   e   f
%
```

Note how we used “[\$R product \$S]”, instead of “\$R product \$S”. The reason is that “\$R product \$S” returns the name of a view command object, not its contents. By adding an extra pair of [], we cause it to dump its contents, just like “\$R” does. We could also have used the following equivalent sequence:

```
% vset x [$R product $S]
% $x
  A   B   C   D   E   F
  --- --- --- --- ---
  a   b   c   b   g   a
  a   b   c   d   a   f
  d   a   f   b   g   a
  d   a   f   d   a   f
  c   b   d   b   g   a
  c   b   d   d   a   f
% unset x
%
```

View operations can be nested at will:

```
% [[[$T project {C D}] minus \
  [$U project {C D}]]
  C   D
  --- -
  e   f
%
```

And lastly, views can be tied to a Metakit data-file:

```
% vset M [mkopen mydata.db]
% $M names
dirs
% [$M sub 0 dirs] names
name parent files
% [[ $M sub 0 dirs] sub 0 files] names
name size date contents
%
```

Here's an example combining much of the above:

```
% vset D [[mkopen mydata.db] sub 0 dirs]
% [[ $D project {parent name}] sort]
parent name
-----
-1 <root>
 0 doc
 0 lib
 2 Class1.0
 2 ClassyTk1.0
 2 Extral2.0
 2 Mpexpr10
 2 Tktable2.7
(etc...)
```

Here is the set of view operators currently available:

*add addcol all as at blocked cmp col cols
concat counts decref delete divide expr first
flatten get groupby if ifmap incref insert
intersect join join0 join1 last mapcol maprow
meta minus names norows nspread omitcol
omitrow pair pick print product project
rename repeat reverse row rowid rows set
single slice sort sortmap spread sub subcat
types union uniqmap unique vid*

The list of operators is still evolving, but as you can see all key relational- and set-operators are included.

Advanced aspects of Ratcl

There is a lot more to say about Ratcl than will fit in this paper. A few highlights:

Calculated fields – data can be generated as a result of calculations based on other fields:

```
% [$T pair [$T expr F:I {B > 'b'}]]
A B C D F
- - - - -
a b c d 0
a b e f 0
b c e f 1
e d c d 1
e d e f 1
a b d e 0
%
```

The current parser is not yet able to handle callbacks, but once this is implemented, arbitrary Tcl-based computations will also be usable inside views.

Derived views are cheap – views are “lazy”, i.e. the information extracted from views is produced on-demand, at the latest possible moment in time. For example, setting up a sorted view is instant, only when rows in it are accessed does the sorting take place. For the same reason, access to views stored on

file can be extremely quick, since only a minimal amount of information is actually read in.

This has profound implications for situations where only a subset of the results is used. One example is the presentation of views on-screen: large views need not be fully accessed when only a small part of the view is showing on the screen.

Sub-views – in contrast to traditional relational database systems, views can be nested. The result of the standard “join” and “groupby” operators is in fact just that: a view with nested sub-views. This greatly simplifies processing, and is dramatically more efficient than producing a result where all data is expanded to fully “flat” tabular form.

The “flatten” operator can be used to force a flat operation when needed, though.

As all other operators, “join” and “groupby” are lazy performers, with everything happening behind the scenes in a totally virtualized manner. This means, for example, that joining two huge views takes little more than two integer vectors of memory, which are set up the moment access to the result is requested.

Cleanup – the view command objects of Ratcl use an elaborate reference counting mechanism to make sure they are kept around as long as needed, but no longer.

The consequence has already been seen in the use of “vset” instead of “set”. The reason for this is that an “unset trace” is needed in Tcl to make sure views are cleaned up when its variable goes away (implicitly on return, in the case of local vars in a procedure).

A somewhat unusual aspect of view command objects is that by themselves they will self-destruct after a single call. This allows the combination of multiple view operations into a single statement, without creating uncollected “debris”. The flip side is the need to use “vset”. This restriction could be lifted if a future version of Tcl were to make the standard “set” just a little smarter, by the way.

Related packages

For reference, here is a brief list of Tcl packages which offer some of the same functionality as Ratcl:

- NAP (“Numeric Array Processor”) by Harvey Davies offers vectorized processing of data. It is geared towards numeric processing whereas Ratcl works equally well with strings.
- TclRAL by Andrew Mangogna is Relational Algebra system that stays very close to the pure relational model, using the “relvar” and “relation” terminology. It is entirely value-based, and as such a good fit for Tcl, but it has no persistence, other than dump/restore.

As has become clear with Metakit over the years, there are very few systems around with relational algebra as basis, and offering the persistence of databases without adopting the SQL language.

The case for Ratcl

Ratcl aims to bridge that gap between databases and Tcl, offering the benefits of both as much as possible.

By “claiming” control over all data, it provides very efficient view “operators” as well as persistence.

The current set of operators is already reasonably complete, but a number of planned improvements will take this even further, such as allowing arbitrary bits of Tcl code inside view expressions – very similar to the way Tcl’s “expr” commands adds an algebraic notation to Tcl while still allowing “[...]” inside any expression to escape back to Tcl.

The central concept is the view, which maps to a Tcl command object – much like widgets map low-level GUI concepts via Tk. Views can be passed around and combined at will. Unlike most commands, views represent lazy evaluation, where the actual processing takes place behind the scenes at various points in time. Setting up complex nested calls to view operators is about preparing for processing, rather than having data handling actually being done.

As a consequence, Ratcl can do a lot of internal optimization, delaying file access and computations until the time they are actually needed. Combined with the column-wise structure of data, this often leads to a substantial reduction of processing time.

The efficiency of views in Ratcl will be presented in the next section.

Size and performance

The Ratcl extension consists of a tiny “core engine” coded in C, a bit of Tcl glue code, and some auxiliary data. A complete system, including all the relational and set operators, a Metakit data file reader, and an expression parser is about 75 Kb. With compression, a standalone exe containing all of the above as well as a Zlib de-compressor ends up being 22 Kb.

The source code of all the pieces of Ratcl amounts to some 3000 lines of code, half of which is C.

Small is beautiful, not just as an academic challenge, but because less code means fewer places for bugs to hide, and fewer cases to deal with and test. The layering used in Ratcl means that the system consists of a small set of carefully chosen components, each highly dedicated and aimed at only performing a few tasks, but doing those real well.

The performance of Ratcl has not been optimized at all so far. Key operations such as join and groupby use algorithms which are far from optimal right now, the reason for this is that this implementation focuses on functionality and took many shortcuts to get the basics working, regardless of overhead.

Nevertheless, Ratcl can open and access Starkits, which are Metakit data files, faster than the Mk4tcl extension itself. In plain integer column iteration, Ratcl can outperform Mk4tcl by a factor 4, in string iteration it is about on par.

In another test, using an Apache log file with about a million entries, it takes 1.66 sec to locate 3 copies of a specific IP address in today’s basic Ratcl (Metakit: 2.18, SQLite: 3.85). All timings are done on a relatively slow PIII/650 notebook to get a decent timer resolution.

The comparison with SQLite is a bit unfair, since one should use an index, in which case the time drops to 0.32 mSec. Then again, note that adding the index took 37 sec, and dropping it again took another 3 sec, so the choice of what to index is an important one to make up front.

To construct a comparable case in Ratcl requires creating a view which projects the key and then sorts it. With sorted data, binary search can then be used to locate a key. In Ratcl, project + sort take about 0.4 sec, and searching takes 28 microseconds).

The conclusion at this point should be that although Ratcl’s brute force is surprisingly efficient, it is no match for indexed access when the number of records involved is large (we’re comparing $O(N)$ brute force with $O(\log N)$ binary search). At this point, similar tricks must be used to gain optimized access, after which a Ratcl-based solution again outperforms other databases by an order of magnitude. Similar results and ratios can be expected with hashing, by the way.

Now, as everyone doing benchmarks knows, it’s fairly easy to “construct” examples that support any type of conclusion. Therefore, in the following discussion all further comparisons have been omitted.

Instead, let’s simply examine how long it takes to perform certain tasks using the high-performance primitives built into Ratcl (but not yet used very much!).

Opening the above data file takes 720 mSec. Using a primitive call, locating 3 ints in a million on file takes 20 mSec (80x as fast as Ratcl’s current dumb code).

One point to make is that most database timings are severely skewed towards single accesses, a metric which is usually irrelevant. What matters, is the performance figures when large amounts of data are processed as a whole. This is where databases can get dogged down to hours of processing time and I/O-bound disk thrashing. This is also where Ratcl’s column-wise model tends to make a dramatic difference.

The above example of finding 3 matching ints in a million takes exactly as much time regardless of the number of results – i.e. 20 mSec to find all values larger than K, for any K.

At the time of writing, not many more performance results are available. As mentioned before, Ratcl does not yet hook into the optimized vector-oriented code that is part of the system – most of the effort so far has simply gone into getting the data structures ready for vectorized use, and implementing basic functionality.

One more result which ought to give an impression of what lies ahead for joins and groupby is available: a hash-based algorithm which identifies all identical values in a set of the same million integers as above, takes 0.15 sec. For comparison, Tcl's "lsort –unique –integer" takes 3.7 sec to produce the same results (about 20,000 groups). Note also that these integers consume 4 Mb memory in Ratcl and 28 Mb in Tcl.

The explanation for these results, which show orders of magnitude higher performance figures than current database systems, is that the combination of an inverted column-wise design with a very efficient data format which is identical on-file and in-memory, work together to take maximum advantage of today's CPUs. Not only is a column-wise structure optimal for file access, it also lets CPU caches work at their best. All it takes is a highly vectorized internal design of the underlying code engine.

Reasons to use SQL

Despite these nice results in Ratcl, there are still a number of reasons to use SQL in an application:

- It's a standard – there is a lot of code based on SQL and a lot of experience with it.
- It's convenient to write tasks in a non-procedural way. The ability to think in terms of *what* instead of *how* is a huge time-saver, even if performance might suffer a bit.
- And lastly: you may not have a choice, if your boss dictates it. The same holds for Tcl itself, of course!

SQL is a complete language of its own (several in fact, sometimes frustratingly so). By adding SQL to an application, you are bound to get more or less of an impedance mismatch – quoting rules change, variable naming and expansion changes, even simple operators change ("<" versus "!=" for example). There is also some duplication of functionality, such as SQL's "like" versus Tcl's "string match". And lastly, you may find that SQL does not offer regular expressions, and that Tcl's "regexp" cannot be used for string searches in data managed by the database.

SQL is a language (from the 60's, in fact) - and its use in Tcl unavoidably implies working with two sometimes very different ways of looking at data.

Even though SQL is quite well standardized, the availability and lack of features differ widely across different database implementations and their bindings to Tcl. There are database independent wrappers and there is ODBC – but be prepared for quite a bit of tinkering. SQL is nice, but definitely no panacea.

Introducing Rasql

Rasql aims to bridge the world of databases and Tcl, but in a very different way than Ratcl.

Rasql is an implementation of SQL, and as such offers the standard SQL notation for those who choose to work this way.

The crucial point to make is that Rasql is based on Ratcl – it is in fact a thin layer over Ratcl, parsing and translating SQL statements to relational algebra operations in Ratcl.

This has a several implications:

- Rasql simply presents itself as an extra set of view operators, the most important one being called "select".
- You can combine views constructed with Ratcl with Rasql's standard SQL syntax.
- Views use the same inverted-column design, and are very efficient in space and time.
- The result of a Rasql "select" is a view.
- There is some usefulness in having sub-views, but there are also some limitations on their use inside SQL, which was not designed for them.

That last note means that Rasql can also be used as basis for further Ratcl operations. So now you get the best of both worlds: use SQL's non-procedural notation when it is convenient, yet switch to view operators as needed.

Rasql is not a gimmick. It handles nested sub-queries and quite advanced cases of SQL. Its design differs fundamentally from most SQL implementations, in that it translates non-procedural requests to set-wise manipulation of data, just as Ratcl does – this takes full advantage of the internal column-wise design.

At least four different implementations of more or less complete SQL engines on top of Metakit have provided the insights needed to accomplish this. Rasql combines this experience and brings it to Ratcl.

Rasql's limitations

One pretty severe limitation of Rasql is that it is work in progress. Its last implementation is from 2004, and was based on a predecessor of Ratcl. This code is not ready for serious use, and needs to be rewritten to use the latest Ratcl code base.

Another limitation of both Ratcl and Rasql right now, is that there is no built-in support for storing NULL. This can be emulated quite efficiently by adding an extra flag to every NULL-able column, but computations with such an approach can become a bit tricky. The reason NULL has not been added yet is that it requires a change to the Metakit file format to allow persisting views where some data items can be NULL.

The use of NULL is extremely controversial in the formal relational database world. Still, to provide sufficiently compatible support for SQL it will need to be supported in Ratcl and Rasql. Sub-views also offer a way to avoid NULLs in join and groupby.

Rasql does not aim to support SQL 100% (if that were even possible). The goal of Rasql is to support enough of the language to perform all common tasks, and to offer as few surprises to people who are used to SQL as possible. Rasql is a gesture towards what has become a de-facto standard, not an endorsement, and certainly not “Yet Another SQL Database”.

Lastly, Ratcl and Rasql are single-process in their current design. A number of high-performance concepts for contention-free parallelism in Metakit will be ported to Ratcl (and hence Rasql), eventually.

Note that this does not mean that Ratcl and Rasql are single-user. Multi-user scenarios will be fully supported as client/server option, once transactions are added back in, with all the aspects of ACID (atomicity, consistency, isolation, and durability) covered.

Current status

Right now (early May 2005), the Ratcl package is about to enter its second public release. This release supports general-purpose views, a wide range of view operators, read-only access to Metakit-compatible data-files, and simple serialization of views to file.

The current performance level of Ratcl is “decent”, meaning it’ll compare just fine with other solutions, but also that it is still far from the intended levels. The reason for this is that a lot of the internal vector-oriented processing has not yet been activated.

This Ratcl release will not be suitable for production use, it’s really a technology preview – to allow others to get more experience with the design and comment on it, and to act as a baseline for optimization.

The stability of Ratcl is already very good, i.e. it does what it should do. Robustness is not quite there yet, i.e. if used incorrectly, Ratcl still falls over far too often to be usable in general.

There is a nice introduction to Ratcl on the web, but it refers to an earlier implementation – some details of the syntax have changed by now. The semantics of it all is largely unchanged, though.

Rasql will not be released in public for some time to come, although the code will be made available as soon as the port to the latest code base is completed.

The Vlerq research project

Ratcl and Rasql are part of a research project called “Vlerq”. Vlerq is an acronym for:

Take Vectors
Add a Language
Make it Embeddable
Use the Relational model
Include a Query mechanism

Ratcl and Rasql are the result of using several tools being developed in / for Vlerq. In particular, a high-performance vectorized virtual machine called Thrive (Threaded Interpreter Vector Engine), and a systems-level language called Thrill (Thrive Language Layer).

The Thrive VM is a very tightly coded stack machine in C with an emphasis on handling vector operations and persistent data with maximum efficiency. Thrive includes automatic garbage collection. The Thrill language is relatively low-level, and is loosely based on Forth and other “concatenative” languages. Most of the Ratcl logic is coded in Thrill.

Much of the expected performance of Ratcl and Rasql are due to the fact that Thrive and Thrill have been designed and implemented from the ground up to provide the necessary functionality. The results so far and the extreme compactness of the code show that by segmenting a project into different conceptual layers (combining C, Thrill, and Tcl), far more can be accomplished than with a single-language design.

In a way, the Vlerq project is really a tribute to John Ousterhout’s vision on scripting as a glue language.

Longer-term goals

The use of views as central mechanism for data exchange is only the beginning of a considerably more ambitious goal: to create a data-flow driven framework whereby processing becomes completely automatic.

The promise of data-flow is that it allows you to move away from “thinking about all the consequences all the time”. Instead of applying changes to data and hard-coding the consequences at each point where such changes are made in an application, data-flow computing provides the same capability as what spreadsheets have been offering for decades.

With data-flow as driving mechanism, there could be a revolution similar to event-driven programming in user interface development, but permeating all the aspects of application development this time around.

To achieve this, the distinction between data on-file and in-memory has to be removed, which is precisely what Ratcl’s “views” are for. This can only be done by “taking the data out of Tcl”, i.e. adopting a coding style whereby Tcl manage dependency structures, but not directly the data itself. This is nothing new: the same holds for GUI components in Tk.

Getting data-flow working “all the way to the GUI” will one day require some new “data aware” widgets. Discussion on this is beyond the scope of this paper.

Conclusions

This paper has presented some early results of Ratcl and Rasql, two packages for Tcl that aim to simplify data manipulation.

As several preliminary tests with Ratcl show, the performance that can be achieved is at least an order of magnitude higher than traditional databases.

The reason for this is that an “inverted” column-wise data structure offers significant benefits for vector-oriented data processing algorithms.

The consequence is that even when not using any auxiliary “indexes”, many tasks will be surprisingly efficient. This means that we can have your cake and eat it too: the flexibility of not having to design rigid data models up front, combined with performance which exceeds most databases, and sometimes even Tcl’s performance with its own data structures.

With Ratcl and Rasql, it becomes feasible to “just start coding”, which is one reason why scripting languages can be so effective. This should of course not be taken as an excuse to design scripted applications badly, or worse, to skip the design phase entirely!

The column-wise format of persistent data makes adding columns trivial and instant, and the very high performance of joins, groupby, and sort means that the usual agony of choosing just the right set of indices and entering SQL statements in just the right order becomes a thing of the past.

What this means is that with data in Ratcl, you can get the best of everything:

- Data structures which are easy to define *and* to later extend or alter.
- Efficient operations on large amounts of data.
- Compact representations in memory and on file.
- Tcl-like performance as well as robust persistence.

Much of this is not new. People programming with APL, J, and K have known for decades that a wide range of processing tasks can be done far more efficiently than is commonly known – and that a vectorized language can be extremely concise yet flexible.

What Ratcl and Rasql bring to the table is the ability to get the best of both worlds. By introducing view command objects as the one generic data structure for everything, and by embedding this very tightly in Tcl, the result is a system in which data manipulation becomes very convenient, avoiding the usual looping idioms and dealing with entire data sets in one step.

Ratcl, and especially Rasql, are still in their infancy. Although all results presented so far are based on working code, that code still is being revised daily.

It is hoped that the main benefits (and trade-offs) of the approach presented here will help others see how the impedance mismatch between traditional database systems and a programming language such as Tcl can be reduced, by using “views” as general-purpose data

structure, combined with relational algebra, set operators, and array operators.

The Vlerq project which has become the foundation of Ratcl and Rasql has its own home page on the web at <http://www.vlerq.org> - a wiki-based area for all discussion and news related to this project.

All software described in this paper is available under the MIT open source software license.

Acknowledgments

I would like to thank Mark Roseman and Steve Landers for the many discussions which led to the design of Ratcl, Rasql, and Vlerq over the years. I would also like to thank them for their help and review of this paper.

Much of the Vlerq architecture stems from the experience gained with the Metakit database library in over a decade. I would like to thank everyone who directly or indirectly helped me refine and improve that system, often simply by pushing for more performance or identifying subtle bugs and design limits.

A very big thank you also to Mike Doyle and Eolas Technologies Inc, for funding the Vlerq project since early 2005, which has allowed me to make very substantial and rapid progress with the Ratcl and Rasql software.

References

Ratcl home – <http://www.equi4.com/ratcl.html>

Rasql technology preview and online demo – <http://www.equi4.com/preview/>

Metakit – embedded database extension for Tcl (Mk4tcl), <http://www.equi4.com/metakit.html>

The Tcl’ers Wiki - a collaborative web site for the Tcl community, <http://wiki.tcl.tk/>

NAP – Numeric Array Processor by Harvey Davies, <http://wiki.tcl.tk/4015/> / <http://tcl-nap.sourceforge.net/>

TclRAL – by Andrew Mangogna, <http://wiki.tcl.tk/12348/> / <http://tclral.sourceforge.net/>